National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

NetGen: A Tool For Partitioning Sequential Programs For Net-Based Execution

by

Daryl James Maier ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical Engineering

Edmonton, Alberta
Fall 1995

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-06505-7

Canadä

University of Alberta

Library Release Form

Name of Author: Daryl James Maier

Title of Thesis: NetGen: A Tool For Partitioning Sequential Programs For Net-Based Execution

Degree: Master of Science

Year this Degree Granted: 1995

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Daryl Maier

Daryl James Maier
4124-125 Street
Edmonton, Alberta, Canada
T6J 2A3

August 31, 1995

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **NetGen: A Tool For Partitioning Sequential Programs For Net-Based Execution** submitted by **Daryl James Maier** in partial fulfillment of the requirements for the degree of Master of Science

Professor W. Joerg (Supervisor)

Jack Mowchenko

Dr. J. Mowchenko

Xiaobo Li

Dr. X. Li

Date: August 28, 1995

Abstract

This thesis describes the development of an experimental tool that allows a researcher to study the partitioning of a sequential program suitable for execution in a net-based computing environment. Net-based computing models the execution of software as a Petri net, where segments of sequential code represent places and transitions coordinate the execution of such segments. This model allows the representation of synchronization, sequentiality, and parallelism between segments of code.

The tool (NetGen) bases its parallelization decisions on the results of a static dependency analysis of a Pascal program. A dependency net structure is produced that summarizes the dependencies between statements and their required execution order. The partitioning process groups (or coalesces) statements in the net that have a particular dependency relationship that the researcher wishes to explore. Statements within these groups are executed sequentially and parallelism can exist between different clusters. Three different methods of grouping nodes are available for investigation in addition to several supplementary net generation parameters.

Acknowledgements

I am grateful for having the opportunity to work under the supervision of Professor Werner Joerg. His ideas and insight have allowed me to learn much about the exciting area of parallel computing. I would also like to thank the members of my examining committee, Dr. Jack Mowchenko and Dr. Xiaobo Li, who took the time to read my thesis on short notice. Their comments and suggestions were invaluable and very much appreciated.

Finally, I would like to thank my parents and family who have provided much encouragement and support throughout my years of study.

Table of Contents

List of Tables
List of Figures
List of Symbols
Chapter 1 Introduction 1
1.1 Motivation 1
1.2 Thesis Goals 4
1.3 Thesis Organization 5

Chapter 2 Parallel Execution of Software 7
2.1 Introduction 7
2.2 Parallel Computing Paradigms 8
  2.2.1 Parallel Computer Architectures 8
  2.2.2 Parallel Programming Paradigms 10
  2.2.3 Net-Based Computing 10
2.3 The Software Parallelization Process 11
2.4 Software Partitioning 13
  2.4.1 Partitioning Correctness 14
  2.4.2 Partitioning Efficiency 14
  2.4.3 Sources of Parallelism in Programs 15
2.5 Software Development Environments 16
  2.5.1 Manual Parallelization 16
  2.5.2 Automatic Parallelization 18
    2.5.2.1 Parafrase-2 20
    2.5.2.2 PTRAN 20
  2.5.3 Semi-Automatic (Interactive) Parallelization 21

Chapter 3 Flow and Dependency Analysis 22
3.1 Introduction 22
3.2 Control Flow Analysis 23
3.3 Data Flow Analysis 24
3.4 Control Dependence 26
3.5 Data Dependence 29
3.6 I/O Dependencies 30
3.7 Representing Dependency Information 31
3.8 Conditions For Parallelism 32

Chapter 4 A Program Partitioner For Net-Based Execution 33
4.1 Introduction 33
4.2 Target Architecture 34
4.3 Sequential Programming Language 35

4.3.1 Language Elements 36
4.3.1.1 Statements 37
4.3.1.2 Blocks, Regions, and Scope 37
4.4 Annotated Net Generation 38
4.4.1 Dependency Annotated Net 39
4.4.1.1 Dependency Edges 44
4.4.1.2 Dependency Regions 45
4.4.1.3 Condition Regions 47
4.4.1.4 Block Depth 47
4.4.2 Intraprocedural Program Analysis 48
4.4.2.1 Parsing The Sequential Program 48
4.4.2.2 Declarations 48
4.4.2.3 Dependency Net Nodes 49
4.4.2.4 Control Dependence 51
4.4.2.5 Analysis Of Multiple Blocks 52
4.4.3 Interprocedural Analysis 53
4.4.3.1 Alias Resolution 53
4.4.3.2 Dependency Computation 54
4.4.4 makenet Invocation Syntax 55
4.5 Node Migration 56
4.5.1 nodemig Invocation Syntax 60

Chapter 5 Program Partitioning Via Node Coalescing 61
5.1 Introduction 61
5.2 Supernodes 62
5.3 Supernode Properties 63
5.4 Supernode Formation Rules 64
5.5 Node Coalescing Techniques 66
5.5.1 Node Coalescing Via A Dependency Strength Threshold 66
5.5.1.1 Dependency Strength Calculation 68
5.5.1.2 Pseudo-Dependency Strengths 69
5.5.1.3 Computing Target Nodes 70
5.5.1.4 Coalescing Threshold 72
5.5.1.5 Node Coalescing Algorithm 72
5.5.2 Coalescing Via Number Of Dependencies Threshold 74
5.5.3 Coalescing For Maximal Node Parallelism 75
5.6 coalesce Invocation Syntax 76
5.7 Net Description Language 77
5.7.1 Language Description 77
5.8 NetGen Implementation 78
5.8.1 Development Tools 78
5.8.2 Correctness Of Design 78
5.8.3 Design Safety 78

Chapter 6 Program Partitioning With NetGen 80
6.1 Introduction 80

6.2 Sample Program analysis 80
6.2.1 Program Description 80
6.2.2 Node Coalescing 80
6.3 Sample Program leastsquares 80
6.3.1 Program Description 80
6.3.2 Node Coalescing 81
6.4 Sample Program Graphics 82
6.4.1 Program Description 82
6.4.2 Node Coalescing 83

Chapter 7 Conclusions and Future Work 89
7.1 Conclusions 89
7.2 Future Work 91
7.2.1 Petri Net Generation From A Coalesced Net 91
7.2.2 Coalescing Supernodes 92
7.2.3 Dependency Analysis Improvements 92
7.2.4 Source Language Improvements 94

Bibliography 96

Appendix A Grammar Descriptions 100
A.1 Pascal Subset Grammar 101
A.2 Net Description Grammar 106

Appendix B Algorithms 109
B.1 Alias Resolution 110
B.2 Dependency Computation 111
B.3 Node Migration 114
B.4 Calculation Of Target Nodes 116
B.5 Coalescing By Dependency Strength 118
B.6 Coalescing By Number Of Dependencies 121
B.7 Coalescing For Maximal Node Parallelism 123

Appendix C Sample Files 125
C.1 Sample Net Description File 125
C.2 Sample Node Coalescing Parameter File 131
C.3 Sample Program analysis 132
C.4 Sample Program LeastSquares 133
C.5 Sample Program graphics 134

List of Tables

Table 3.1 Control dependencies for the sample program in Figure 3.1a **28**
Table 4.1 Types of nodes available in the dependency annotated net **39**
Table 4.2 Data retained by nodes in a dependency annotated net **50**
Table 5.1 Pseudo-dependency parameters **70**
Table 6.1 Program analysis Coalescing Via Dependency Strength (CR = 1.0) **81**
Table 6.2 Program analysis Coalescing Via Number Of Dependencies (ND = 1.0) **81**
Table 6.3 Program analysis Coalescing For Maximal Node Parallelism **82**
Table 6.4 Program leastsquares Coalescing Via Dependency Strength (CT = 1.0) **83**
Table 6.5 Program leastsquares Coalescing Via Dependency Strength (CT = 1.0, PSEUDO_CONTROL_T = 1.5) **83**
Table 6.6 Program leastsquares Coalescing Via Number Of Dependencies (ND = 1.0) **84**
Table 6.7 Program leastsquares Coalescing Via Number Of Dependencies (ND = 2.0) **84**
Table 6.8 Program leastsquares Coalescing For Maximal Node Parallelism **85**
Table 6.9 Program graphics Coalescing Via Dependency Strength (CT = 1.0) **86**
Table 6.10 Program graphics Coalescing Via Number Of Dependencies (ND = 2.0) **87**
Table 6.11 Program graphics Coalescing For Maximal Node Parallelism **88**
Table B.1 Algorithm symbols and convenience functions **109**

List of Figures

Figure 2.1 Petri net constructs **10**
Figure 2.2 The program parallelization process **12**
Figure 3.1 Example control flow graph **24**
Figure 3.2 Aliasing of variables via procedure parameters **27**
Figure 3.3 Data dependencies between statements **30**
Figure 4.1 NetGen's processing steps required to produce a net-based representation of a sequential program **33**
Figure 4.2 Execution environment for a net-based program **35**
Figure 4.3 makenet processing steps **38**
Figure 4.4 Demonstration of unnecessary dependencies **41**
Figure 4.5 The mapping of a program into execution steps **42**
Figure 4.6 Mapping Pascal statements into a dependency net **43**
Figure 4.7 Hard and soft dependencies in the annotated net for the sample program in Figure 3.1 **45**
Figure 4.8 Block depth, dependency regions, and condition regions **47**
Figure 4.9 Determination of dependencies between nodes in different dependency regions during node migration **58**
Figure 4.10 Node migration **59**
Figure 5.1 Input/output requirements of the node coalescing phase **62**
Figure 5.2 Node coalescing violating a necessary dependency **65**
Figure 5.3 Example of inefficient node coalescing **66**
Figure 5.4 Calculation of dependency strength between statements **69**
Figure 5.5 Node coalescing using dependency strength criterion **72**
Figure 5.6 Node coalescing for maximal node parallelism **76**
Figure 7.1 Conversion of an annotated net into a Petri net for net-based execution **93**

List of Symbols

$\emptyset$	empty set
$\in$	is an element of
$\notin$	is not an element of
$\ni$	such that
$\wedge$	logical and
$\vee$	logical or
$|S|$	cardinality of set S

Chapter 1

Introduction

This thesis describes NetGen, an experimental tool for partitioning Pascal programs into subtasks for execution on net-based multiprocessor computers [MaJo95]. The partitioning process is guided by the identification of segments of code that could potentially be executed in parallel. Several techniques for forming and clustering code segments are available to the researcher. The resulting parallelized program is stored in a format suitable for conversion to a Petri net representation that can be executed on the target computer.

The work is part of a greater research effort exploring experimental multiprocessor architectures and parallel software development in the Department of Electrical Engineering at the University of Alberta.

1.1 Motivation

It is accepted knowledge that the execution of a program as multiple, parallel subtasks can potentially reduce its overall execution time. Parallel processing has received considerable attention of late as the physical limits of a single processor are fast approaching. As the performance requirements of today's applications continue to increase, parallel processing is perceived as an attractive solution for potentially increasing performance using existing technology.

Unfortunately, the performance benefits of parallel processing come at a price. An execution speedup is not guaranteed, and in fact depends on the application type. Scientific applications that perform numerous repetitive calculations are often the best candidates for parallelism, whereas general-purpose applications do not usually exhibit a great deal of useful parallelism. End users, however, can only benefit from such performance gains if appropriate hardware is available and if the software has been written or "massaged" for parallel processing.

In general, two opposite approaches can be taken when developing software for parallel computers: manual or automatic parallelization. The former approach is language oriented, leading to dedicated parallel languages (e.g., Ada, Occam), to language extensions fitting particular classes of machines (e.g., Parallel Fortran, Concurrent Pascal), or to operating system extensions for inclusion in sequential languages (e.g., Enterprise system [Wong92]). Presently, the latter approach is mainly focused on particular machine architectures; it has been extensively investigated in scientific computing for vector machines, and to a much lesser extent for dataflow machines. However, programming parallel machines is an arduous, and by no means intuitive, task. In addition, a tremendous amount of tested and reliable software already exists for sequential computers. Consequently, automatic parallelization may prove a more efficient and productive way to exploit the hardware potential.

To make multiprocessing computers more appealing to a wider range of users they must be made cost effective. At present, to attain this goal requires that parallel processing platforms be built based on multicomputer architectures. Unfortunately, the type of parallelism that can be readily exploited on such architectures is usually coarse-grained. Finer granularities require more expensive and special-purpose hardware, found mostly in technologically immature dataflow machines.

To obtain a cost effective approach with potential performance improvement, a hybrid approach, combining promising dataflow parallelism with the cost-effectiveness of realistic multicomputer systems, is studied. The hybrid architecture is an abstract machine based on the Petri net concept. Presently, it is being used as an intermediate notation between two main problem areas: the investigation of ways to generate net representations from sequential programs and the research of low-cost architectures best suited for parallel execution of programs represented as Petri nets.

The two problem areas are interdependent. The development of net-based architectures is guided (in part) by a measure of the amount of parallelism that can typically be extracted from general-purpose programs. Similarly, the translation of programs into nets is influenced by the architectural constraints of the target computer. Therefore, a tool is required that allows a researcher to examine how the parallelism in any arbitrary program can be formed into a net, governed by certain architectural constraints. These constraints appear as net generation parameters that can be modified such that the effects on the resulting net structures can be studied. For example, one such parameter could represent the cost of a context switch between parallel subtasks. Consequently, the structure of the nets formed should be an indication of how the general-purpose parallelism would be exploited on a particular target computer.

The research direction taken by this project is also motivated by two recent contrary viewpoints on the status of parallel processing in modern computing. On one hand, [Lewis94] claims the future of parallel computing lies on the desktop with "multiprocessor systems consisting of four, eight, and 16 processors". Traditional mainframe and massively parallel multiprocessors are far too expensive to be practical for widespread and general-purpose use. Consequently, smaller, less expensive configurations that contain fast sequential processors appear much more attractive for exploiting parallelism within the applications run by the ordinary user. The impact of parallel processing technology is highly dependent on the development of effective parallel software languages, development tools, and computing environments. The ultimate goal of the parallel computing research initiative at the University of Alberta is to develop a machine that can easily exploit the parallel properties of general-purpose programs and to do so with the user being oblivious to the process. The NetGen project attempts to make a positive contribution to parallel computing by providing a tool that can be used to explore ways of mapping general-purpose applications to parallel machines. It is also hoped that this tool can be used to influence the development of parallel architectures that are suitable for exploiting the parallelism characteristics found in any arbitrary program.

The second viewpoint provides a much more cynical outlook on the present status and future of parallel computing. [Furht94] claims that parallel computing is no longer viable and has few applications in modern computing. He envisions no future for multiprocessing and is appalled by the amount of research devoted to the area. Furht further declares that graduate students should instead concentrate their research efforts on "hot" areas that will guarantee them jobs in today's marketplace. This is a very near-sighted outlook on a topic that many researchers believe contains several unexplored areas for research. While it is true that parallel computing has stagnated of late, this is certainly not an indication that there are no research avenues remaining to be studied. On the contrary, perhaps this indicates that a new perspective needs to be taken on parallel computing. Instead of concentrating on developing traditional architectures and paradigms, perhaps new computing paradigms and different technologies should be explored (e.g., non von-Neumann machines and dataflow ideas). The net-based parallel computing paradigm studied in this project is one such novel area of research. As its exploration is only in its infancy, it would be immature to claim that research in this direction is impractical. At present, suitable tools need to be created to make the determination of the effectiveness of this paradigm. NetGen represents only the first step in this process and should prove to be a useful tool for experimenting with the net-based programming paradigm.

1.2 Thesis Goals

This thesis shall focus on the translation of sequential programs into a net representation and describe, in particular, a tool for experimenting with net formation parameters. The NetGen project has three primary objectives:

1. To develop a tool that can analyze a sequential program, identify potential areas for parallelism, and then translate it into a representation suitable for conversion into a Petri net notation for execution on the abstract machine.

2. To provide several parameters that can be varied such that the effects on the resulting net structures produced can be studied. This will allow a researcher to investigate the effects of applying the various net generation parameters on general-purpose sequential programs. Ideally, the target machine would be able to exploit all the parallelism that exists within the subject program. Unfortunately, this rarely

occurs because, in general, the parallelism requirements of sequential programs cannot be entirely fulfilled by a single architecture. However, it would be useful if values could be found for the net generation parameters such that any arbitrary program could be executed on a particular machine with an expected performance improvement.

3. It is hoped that the results obtained from NetGen can be used to influence the development of net-based multiprocessor computers and/or the program transformation tools required to execute a program on such machines.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides relevant background material on software development for parallel computers. Some important issues that must be addressed by the developer are discussed along with several examples of parallelization tools.

Chapter 3 discusses program flow and dependency analysis. This type of analysis is essential for any useful automatic parallelizer. Required flow and dependency analysis concepts and pertinent definitions are presented and discussed in the context of the NetGen project.

Chapter 4 describes the NetGen project as a whole but concentrates on a discussion of the initial net generation and node migration phases.

Chapter 5 describes the node coalescing phase which begins to partition the software into tasks. The net description language is described and a perspective on some implementation issues is presented.

Chapter 6 contains the results of partitioning some sample programs using the node coalescing techniques described in Chapter 5.

Chapter 7 describes the conclusions and possible areas for future research with this project.

The Appendices contain sample files, grammar descriptions, and the important algorithms discussed in this thesis.

Chapter 2

Parallel Execution of Software

2.1 Introduction

Computers with more than a single processing element (PE) can potentially decrease the execution time of an algorithm or application. To take advantage of the benefits of parallel processing requires that an application be "partitioned" across all or some of the PE's on the target computer. The performance gains are largely dependent on the particular problem and on how well it has been partitioned.

The partitioning process results in the decomposition of a program into many fragments, called subtasks[1]. The identification of subtasks can be done manually (by the programmer), automatically (by the compiler), or by a combination of the two. Each technique has its merits and shortcomings; all share the common goal of easing the complex task of programming parallel computers.

This chapter will explore some traditional techniques used in developing and partitioning software for parallel computers. A popular parallel computing paradigm will be introduced first, followed by a discussion of multiprocessor architectures and programming techniques that adhere to the model. The software parallelization process is described next, with specific attention focused on program partitioning issues. Finally, proven techniques used to alleviate the complex task of programming parallel computers are described. In the course of this discussion, several parallel software development research initiatives will be presented.

---

[1] sometimes loosely referred to as *processes* or even *proclets* [Ottenstein85].

2.2 Parallel Computing Paradigms

A paradigm is "an illustrative model or example, which by extension provides a way of organizing information." [Budd91] Paradigms are helpful in conceptualizing a problem by relating its solution to the model. Parallel computing is a diverse field with many areas of active research. As a result, it is useful to categorize these research efforts for comparison and performance evaluation purposes using a common, underlying paradigm.

The most popular paradigm for parallel computing characterizes computing environments as either synchronous or asynchronous [LeEl92, AmBuZi92]. Synchronous parallel computing performs identical operations on data in parallel. The operations are arranged such that the dependencies between computations are eliminated. Conversely, asynchronous parallel computing involves parallel processes independently coordinating their activities. The data and instructions executed may differ between individual processes.

This paradigm can be conveniently used to classify both parallel architectures and parallel programming methods. The primary benefit of a mutual paradigm is that it allows a programmer to easily ascertain the appropriateness of a programming language for a particular parallel architecture.

2.2.1 Parallel Computer Architectures

Traditionally, Flynn's hardware taxonomy [Flynn72] was used to classify computer architectures. This scheme organizes architectures into four categories based on the number of instruction and data streams. Although suitable for the simpler architectures of the past, many new parallel architectures do not fall neatly into a single category. Although it has been criticized for its rigidity, it does provide a convenient "shorthand for characterizing [parallel] architectures" [Duncan90] and is a common basis for new classification schemes.

A parallel computing paradigm based on that described in the previous section can be used to conceptualize different parallel architecture designs. [Duncan90] categorizes parallel architectures as being either fundamentally synchronous or asynchronous. This scheme is a superset of Flynn's hardware taxonomy and can accommodate more recent parallel architectures.

Synchronous architectures employ a central controller to coordinate the activities of each processor. All processors typically perform identical operations on different sets of data in a lockstep fashion. These architectures can be further classified into vector, systolic, and SIMD (single instruction, multiple data) categories. Synchronous architectures are amenable to the parallelism found in many scientific problems. Although they can provide significant performance gains in certain problem domains, they do not exhibit the general-purpose parallelism characteristics that are being explored in this thesis.

[Duncan90] bases the classification of asynchronous architectures on Flynn's MIMD (multiple instruction, multiple data) category. MIMD architectures are characterized by a number of processors that can execute independent instruction streams. The underlying architecture is asynchronous, relegating the responsibility of subtask synchronization to the software. In contrast to synchronous architectures, MIMD architectures do not employ a centralized controller for coordinating activities. As a result, individual processors in the MIMD model are autonomous and can be arranged in several topologies. These features make MIMD architectures suitable for exploiting the general-purpose parallelism that can be extracted from any arbitrary sequential program. It is for this reason that an asynchronous architecture is assumed the target for the parallelization tool described in this thesis.

The MIMD model is fairly broad and encompasses many parallel computer designs. It can be further refined to provide additional insight into the interprocessor communication medium: shared or distributed memory. Processors within a shared memory model coordinate their efforts through a global memory that is accessible to all processors. Distributed memory architectures have a separate, private memory for each processor. Communication between processors is accomplished by passing messages through the processor interconnection network. In general, shared memory architectures employ a small number of processors, have relatively low communication overheads, and are not readily scalable. This is due to the increased memory contention with additional processors. Distributed memory architectures are scalable and possibly composed of heterogenous elements, but suffer from communication latencies.

2.2.2 Parallel Programming Paradigms

The asynchronous/synchronous paradigm can also be applied to parallel programming. Shared memory and message-passing models are the most common manifestations of the asynchronous paradigm in parallel programming. The shared address space approach allows a programmer to view a program as a "collection of processes accessing a central pool of shared variables" [KGGK94]. The message-passing approach, on the other hand, models a program as a collection of processes that communicate by sending and receiving messages. It is clear how the two programming approaches correlate to the asynchronous shared memory and message-passing architecture models.

2.2.3 Net-Based Computing

Net-based computing is an alternative method of modelling software based on the asynchronous, message-passing paradigm described above. This approach models the execution of software as a Petri net[1]. Petri nets are asynchronous and contain constructs for naturally representing sequentiality, parallelism, and synchronization of processes. These constructs are illustrated in Figure 2.1. Conveniently, these constructs are essential for describing the parallel execution of software and, as a result, *net-based computing* is the representation of a parallel program as a Petri net.

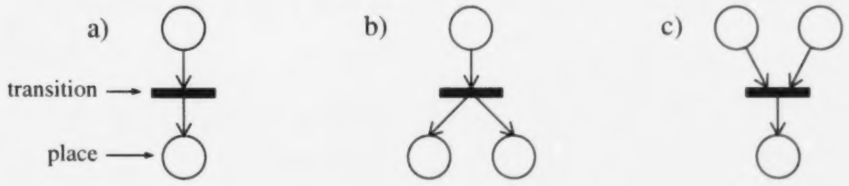


Figure 2.1 Petri net constructs (a) sequentiality, (b) parallelism, and (c) synchronization.

---

[1] For a more in-depth introduction to Petri nets, the reader is referred to [Murata89].

In the net-based computing model, Petri net *places* represent segments of sequential code. The presence of *tokens* in places indicate the execution of the corresponding code segment. Execution of code segments is coordinated by *transitions* in the Petri net.

Although Petri nets may be a natural way of representing parallelism, the process of translating a program into a Petri net format remains an open problem. Existing parallel programs can be mapped rather easily into Petri nets since the program has already been partitioned for parallelism with explicit process synchronization and communication mechanisms inserted. The translation of sequential programs into a Petri net representation is much more difficult. The sequential segments of code that will execute in the Petri net places must be identified and coordinated explicitly. The selection of sequential code segments is by no means obvious and the user would greatly benefit from a tool that could perform this selection automatically.

2.3 The Software Parallelization Process

The execution of software on parallel computers is considerably more complex than on conventional sequential machines. For single processor systems, a program merely has to be compiled into the native assembly language of the target processor and it is ready to execute. The most difficult task is efficiently optimizing a program before it is executed—often the responsibility of the compiler. For multiprocessor systems, the execution of a program is a several stage process. Software must be efficiently partitioned into concurrent subtasks which are synchronized and scheduled onto the processors of the target parallel computer. A program parallelization is often guided by both programmer input and information inferred from an analysis of the software by a compiler. The typical parallelization process is illustrated in Figure 2.2. Certain implementations may insert additional stages or merely concentrate on a particular stage of the process (as in NetGen's case).

The term *parallelization* implies the translation of a non-parallel program or problem into an equivalent parallel program. It does not, however, specify the *method* in which the transformation is to occur. Parallelism within programs can be specified explicitly by the programmer or can be identified using largely automatic techniques by a compiler. With manual techniques, the majority of the parallelization process is the responsibility of the programmer; process scheduling and processor allocation is usually handled by the compiler and/or operating system. Conversely, automatic parallelization is performed largely by the compiler with little programmer intervention.

Automatic parallelization begins with an analysis of the source program to identify all control and data dependencies. Often, many of these dependencies can be resolved by applying special dependency elimination transformations to the source program. These include performing an analysis of subscript variables for arrays, removing storage-related dependencies, and transforming loops to eliminate loop-carried dependencies [Sarkar91a]. Program dependency analysis is machine-independent; the source is examined without considering any properties of the target parallel computer. Chapter 3 is devoted entirely to a discussion of the theory of program flow and dependency analysis pertinent to the NetGen project.

The information inferred from the program dependency analysis is used to guide the identification of subtasks that could potentially be executed in parallel. This is called *program partitioning* and it typically relies on architecture-specific information to obtain a suitable parallelization. Partitioning is often intertwined with process scheduling.

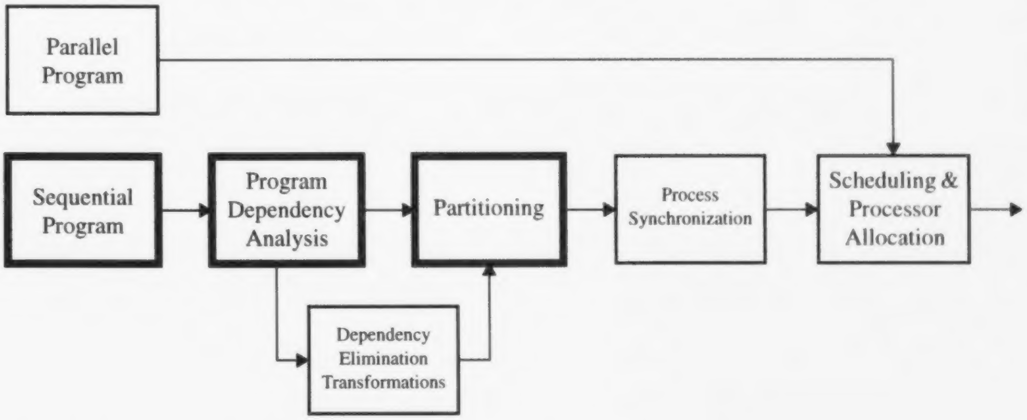

Parallel Program
Sequential Program
Program Dependency Analysis
Partitioning
Process Synchronization
Scheduling & Processor Allocation
Dependency Elimination Transformations

Figure 2.2 The program parallelization process. The emphasized blocks indicate the stages where NetGen concentrates its analysis.

Once the problem has been partitioned, the activities of the subtasks must be *synchronized*, or coordinated. Synchronization is used to ensure a correct sequencing of tasks and the proper use of shared resources. Typical process synchronization schemes include semaphores and barriers and require specialized support from the underlying hardware and operating system. The subtasks must then be optimally *scheduled* on the processors of the target machine to ensure the shortest execution time of the program.

This thesis focuses primarily on the dependency analysis and partitioning phases of the parallelization process. The remaining stages were excluded partly because they are highly dependent on the characteristics of the target machine. One of the mandates of this thesis is to provide a tool that can collect information useful in the search for architectures that can execute net-based programs. As a result, NetGen's analysis is geared toward a particular class of parallel architectures, not a specific parallel machine with well-defined characteristics.

2.4 Software Partitioning

Program partitioning is the decomposition of a sequential program into parallel subtasks. It involves identifying clusters of statements that should be executed sequentially and then determining parallelism between clusters. This process is guided by a program dependency analysis supplemented with architectural information. Due to resource limitations on the target machine, the maximum amount of parallelism within a program cannot always be adequately exploited. Partitioning provides a method of mapping this parallelism to the realistic capabilities of the target computer.

[Sarkar91a] distinguishes two program partitioning strategies: data partitioning and control partitioning. The former approach attempts to minimize communication overhead and improve data locality by partitioning data across subtasks. The latter method attempts to balance the available parallelism and reduce the overhead by partitioning a program into parallel subtasks. Although both approaches are dependent on the target machine, recent partitioning initiatives employ a combination of the two for increased performance. Control partitioning is the method employed by the majority of parallelization tools studied in this thesis. Hence, for the remainder of this work a control partitioning approach will be assumed.

Two new issues arise when developing software for parallel machines that must be addressed: the correctness of the program partitioning and the efficiency of the resulting parallelized code. Although correctness and efficiency are objectives of even sequential programs, they assume additional importance in a concurrent environment.

2.4.1 Partitioning Correctness

When programs are partitioned, it is imperative that the subtasks to be executed in parallel are indeed independent. If this condition were violated then a correct execution of the program cannot be assumed. Thus, an exact determination of all dependencies between potential concurrent subtasks is essential. Of equal importance is the proper sequencing of subtasks once any dependencies have been identified. Concurrent processes must be synchronized such that all dependencies between them are satisfied.

2.4.2 Partitioning Efficiency

The efficiency of the partitioned program is another important consideration. If the overhead of allocating, executing, and reallocating two tasks concurrently is greater than executing the tasks sequentially, then no benefit will be gained from a concurrent execution. An important factor in this issue is that of task size: tasks that are too small may experience problems with communication overhead, while tasks that are too large may contain additional areas suitable to parallelization. Consequently, some measure of the efficiency of the partitioning decisions must be made before program parallelization is considered complete.

A closely related issue is that of process *grain size* or *granularity*. Grain size is a measure of the average number of instructions or statements that constitute a parallel task. Granularity is dependent on the underlying architecture of the target computer and it strongly influences parallel program performance.

[Stone90] defines task granularity as:

> the ratio R/C, where R is the length of a run-time quantum and C is the length of the communications overhead produced by that quantum.

This ratio illustrates the interdependence between task size and machine architecture. Lower ratio values indicate a stronger communication overhead influence on the allocation of parallel tasks and can preclude the use of parallelism altogether. Higher ratio values reflect an attenuation of the effect of communication costs on parallelization and can indicate that parallelism is beneficial.

Coarse-grained parallelism is characterized by a high task granularity. Tasks are large to allay communication costs but the amount of parallelism exploited is usually low. Conversely, fine-grained parallelism is characterized by a low task granularity and a large number of parallel tasks.

Consequently, a trade-off exists between parallelism (small grain size) and communication (large grain size) [LeEl92]. Optimal grain size selection is a process scheduling problem that is known to be NP-complete [Ullman75]. The granularity of a subtask is therefore chosen based on a combination of heuristics and target architecture capabilities.

2.4.3 Sources of Parallelism in Programs

Parallelism can be derived from many sources and can be exploited at many levels. Consequently, the partitioning of a program is not unique. Factors such as the parallelism identification techniques applied, the rigorousness of the dependency analysis, and even the capabilities of the target machine play an important role in the resulting program partitioning.

Perhaps the most obvious source of parallelism lies within the problem to be solved. This type of parallelism is available only to the application developer during the problem decomposition phase. Many scientific problems are inherently parallel. The choice of an appropriate parallel algorithm, implementation language, and data structures amenable to parallelism are also important [Poly88].

Parallelism at the procedure level is considered to be *coarse-grained*. For this type of parallelism to be feasible, subroutines should be independent. This implies that subroutines should represent largely independent tasks that are usually apparent from the problem to be solved. Parallelism at this level is usually structured, which allows the programmer to manually identify opportunities for parallelism.

Loops can be a rich source of parallelism within subroutines. It has been shown that 80% of the time spent in program execution is spent on 20% of the code [Carling88]. Consequently, considerable research has been directed toward identifying parallelism within loops. If it can be determined that different loop iterations are independent then they can be executed in parallel. Parallelism across loop iterations is considered *medium-grained*.

The search for parallelism at the basic block, statement, or processor instruction level is usually considered *fine-grained* parallelism. Parallelism at this granularity is very unstructured and opportunities for parallelism are best determined by a compiler. Usually, special fine-grained architectures (e.g., dataflow) or processors (e.g., VLIW) are required to exploit parallelism at this level.

2.5 Software Development Environments

2.5.1 Manual Parallelization

Parallel programming languages allow the programmer to have the greatest control over partitioning programs for parallel computers. The developer assumes the responsibility of explicitly creating and destroying parallel processes and synchronizing their activities.

The primary advantage of this method of programming is that the developer has a deeper insight into the problem being transformed and can therefore identify, from a problem decomposition perspective, tasks that can be executed in parallel. As a result, manual parallelization is attractive when the problem to be solved is inherently parallel or if greater control is required when specifying parallel tasks.

There are several disadvantages to having the programmer explicitly specify parallelism in an application. Most notably, identifying parallelism within a problem and then coordinating the resultant parallel tasks is an arduous and error-prone process. It is certainly possible that the programmer could create tasks that race or deadlock. Part of the reason for this is an incomplete or incorrect identification of all global data and control dependencies beforehand. This problem will be discussed extensively in Chapter 3. Although knowledge of the problem domain is useful for parallelizing, opportunities for parallelism could easily be missed within the identified parallel tasks.

[Ottenstein85] identifies additional shortcomings of explicitly parallel programming techniques. He indicates that maintenance of parallel programs could be difficult since a program may be modified such that the original explicit process partitioning is no longer as good as it could be. Portability is another issue with such languages. Efficient program partitioning on one parallel computer does not guarantee that execution on a computer with different synchronization and communication overheads will also be appropriate or efficient.

Both parallel programming languages and graphical development tools are available that permit the manual specification of parallelism. The parallel programming language approach is well-established and researched. Recent, widespread availability of graphical workstations, however, has spawned the development of the highly visual and more intuitive graphical development tools.

Most parallel languages are based on extensions to existing sequential languages. Unfortunately, this has led to a large number of dialects for some languages (e.g., Fortran). An essential property of all parallel programming languages is that they contain explicit constructs for specifying parallel process creation, termination, synchronization, and communication. Although the actual constructs vary from language to language, their functions are similar. In general, parallel tasks are created and terminated with some variation of FORK/JOIN statements [AlGo89, HwBr84]. FORK essentially creates a new process that simultaneously executes with the original process. The new process begins executing at the point specified by the FORK statement. JOIN statements are used to terminate parallel tasks by waiting for a previously FORKed process to terminate. Although FORK/JOIN constructs are very powerful, they can be difficult to coordinate and are prone to programmer error. As a result, more controlled constructs such as PARBEGIN/PAREND and PARDO are used. The PARBEGIN/PAREND pair enclose statements that can all be executed in parallel. PARDO is used as a loop header (much like a Fortran DO loop) to indicate that each loop iteration is independent and can be executed concurrently with the others.

Graphical parallelism specification tools are becoming increasingly available. Typically, these tools allow the user to "draw" the processes to parallelize and specify their synchronization requirements graphically. For example, Enterprise [Wong92] allows parallelization of existing C programs by graphically associating subroutine definitions to different icons. The parallelism in a program is specified by the icon types and their interconnections. Some environments integrate graphical parallelization tools into the software life cycle to facilitate the translation of a problem specification directly into a working parallel application [LeEl92].

2.5.2 Automatic Parallelization

An opposite approach to partitioning software involves having a compiler make all parallelization decisions. Automatic parallelization methods attempt to identify parallelism within code that has already been written; they cannot use any knowledge about the problem domain to guide the parallelism determination.

This approach offers several benefits. The primary advantage is that it performs an effortless conversion of an existing sequential program to a parallel platform. Given the abundance of software that has been developed for sequential machines over the last forty years (referred to as "dusty deck" programs [AlGo89]), this approach offers a quick, cost-effective solution to reaping the benefits of parallel computing.

Automatic parallelization does not require the programmer to have any familiarity with the internal details of the software to be converted, nor does it require any knowledge of the parallelization procedure. The techniques used in automatic parallelization are provably correct. As many programs typically consist of thousands of lines of code, this method not only greatly simplifies the conversion process but it eliminates the possibility of a costly programmer conversion error.

Software development for parallel environments has been hampered by the difficulty in debugging parallel software. When dealing with multiple tasks executing concurrently, not only must the program logic within each task be correct, but the synchronization and communication between them must be as well. For this reason, debugging of parallel programs requires a more dynamic analysis than their sequential counterparts. Although parallel debugger technology has improved of late, error detection is still an arduous process [LeEl92]. Until that technology has matured, [ABKP86] suggest developing and debugging software on a sequential machine and then using an automatic parallelization tool to transport it to a parallel platform.

It has also been suggested that automatic parallelization tools are essential at levels where parallelism becomes unstructured [Poly88]. Unstructured parallelism involves parallelism among different instruction and/or data streams and is difficult to identify manually. Such parallelism is typically encountered when trying to exploit finer granularities.

Despite these significant advantages, automatic parallelization suffers from making parallelization decisions based on a static program analysis. Exact determination of all dependencies at compile-time is not always possible, and the parallelizer is forced to employ conservative assumptions about dependencies. As a result, it is possible that dependencies are assumed where none actually exist. These are appropriately referred to as *false dependencies*. They typically arise when the value of a variable cannot be determined at compile-time, when pointers are used, or at procedure call sites. This is the most serious problem with automatic parallelization as the compiler does not have sufficient insight into the program and its characteristics. This important issue is addressed in more detail in the next chapter.

2.5.2.1 Parafrase-2

The Parafrase-2 compiler system [PGHLLS89, ZiCh90] was the first program parallelization tool targeted toward several architectures. It is primarily a research tool used to experiment with the application of program transformations and parallel compiling techniques. The Parafrase-2 compiler is a source-to-source translator that supports Fortran, C, and Pascal programs. Several parallelization transformations are available. These are referred to as *passes* of the sequential code. Passes can be categorized as either machine independent (e.g., conventional dependency analysis), architecture specific (e.g., control dependence elimination on vector machines), or machine dependent (e.g., optimization of register usage). The order in which passes are applied to the program is specified in a *pass list*, which is input to the compiler with the sequential code. Passes are designed to be highly configurable for different parallel architectures and new passes can be developed rather easily. Different passes can communicate via *switches*, which are similar to local and global variables. Estimates of the execution times of segments of code are made at compile-time using the Static Program Analyzer module and this information is used to guide some of the program transformations.

2.5.2.2 PTRAN

PTRAN (Parallel TRANslation) automatically restructures sequential Fortran programs for execution on parallel machines. The compiler attempts to determine the amount of parallelism within a program that can be best exploited on the architecture at hand. PTRAN performs an extremely thorough analysis of the source program. It begins by constructing a variation of the control dependence graph supplemented with data dependence information. This graph is then partitioned into tasks such that all nodes in the task execute sequentially, but the tasks themselves can execute concurrently.

The goal of PTRAN's analysis is to minimize the overall execution time of the program.

To this end, statistics are employed to estimate the expected execution time for a particular partitioning. The average execution frequency of segments of code is determined from execution profiling information generated from previous uses of the software. A task tree is formed by partitioning and merging adjacent tasks based on a minimum execution time criterion.

2.5.3 Semi-Automatic (Interactive) Parallelization

The amount of information that can be extracted automatically from a program is limited by a lack of insight into actual program dependencies. This missing knowledge can be supplied via semi-automatic (or interactive) parallelization tools. Most modern parallelization tools allow some method of removing false dependencies. These range from the user providing compiler pragmas to specify dependency information (called *assertions*) [ZiCh90] to tools that allow graphical display and modification of dependency nets [Harrison90, LeEl92]. Other interactive tools, such as ParaScope [CHHK*93], provide additional capabilities. These include an evaluation of the suitability for parallelism in certain areas of code and an incremental tool for applying parallel transformations.

These tools suffer from many of the same drawbacks as manual parallelization techniques. The compiler will relax some of the dependencies based on the information supplied by the programmer. If this information is incorrect then subtle errors could be introduced into the parallelized program. Unfortunately, this method also requires the user to have some familiarity with the internal details of the software and its expected usage.

Chapter 3

Flow and Dependency Analysis

3.1 Introduction

Automatic and user-assisted parallelization tools offer many challenges to the tool developer. The primary difficulty in developing a parallelization tool for an imperative language such as Pascal is that all dependencies between statements must be identified before the analysis can proceed. Imperative languages are intrinsically sequential and rely on the propagation of computation side effects to produce a desired result. Since the result of executing a statement is dependent on the machine's state prior to its execution, the relative order in which a pair of statements are executed is important [Baldwin87]. To recognize parallelism in imperative languages requires the identification of data and control dependencies in the subject program. Parallelism between two statements is identified by an absence of control and data dependencies between them. Consequently, the amount of parallelism that can be exploited is limited by the number of dependencies that can be safely identified.

The majority of parallelization tools perform a static analysis of the subject program to identify potential areas for parallelization. A static analysis is used to determine as much information as possible about the program at compile-time without requiring an execution of the program. The information inferred from the analysis is applicable to any possible execution of the subject program [MuJo81].

Scalar analysis [ZiCh90] is an application of static analysis useful in parallelizing compilers. It is used to determine scalar variable usage throughout a program. Scalar variables are simply defined as those variables that are not arrays. Arrays may be considered in a scalar analysis, but an access to any element of an array is treated as an access to the entire array. Scalar analysis is an essential phase in parallelization tools and the information inferred is used extensively in the later dependency analysis stages.

Performing a scalar analysis requires an analysis of the flow of control and data through a program. There are three domains in which flow analysis techniques can be applied: local, intraprocedural, and interprocedural. A local analysis typically allows flow analysis to be performed on basic blocks (sequential segments of code with a single entry and single exit point) or on individual statements [ZiCh90, AhSeUl86]. An intraprocedural analysis expands the scope of the flow analysis to an entire procedure. Interprocedural analysis is a further enhancement, allowing flow analysis to be performed across procedure boundaries.

Most parallelization tools perform an interprocedural flow analysis to more accurately determine the flow of control and data throughout a program. However, sometimes only a local analysis is preferred. For example, the Bulldog compiler is an instruction-level compiler that assembles Fortran programs for VLIW (Very Long Instruction Word) machines. For VLIW processors, each instruction contains multiple independent operations. The independent operations are determined by the compiler and are usually found in close proximity to one another in the original source program. As a result, a local flow analysis, concentrating on basic blocks, is performed to achieve the best results [Ellis87, Johnson91].

3.2 Control Flow Analysis

A control flow analysis determines all possible control paths in a program. It is an essential prerequisite for control dependence analysis. Control flow information is conveniently represented in a control flow graph (CFG) and a separate CFG is usually constructed for each procedure in the program. The definition of a CFG in this work is based on [FeOtWa87].

Definition 3.1: Control Flow Graph

A *control flow graph* (CFG) is a directed graph G augmented with a unique entry node BEGIN and a unique exit node END such that each node in the graph has at most two successors. Edges between nodes represent transfers of control between nodes. Conditional execution is indicated with T (true) and F (false) edge labels whereas unconditional execution is indicated by the absence of an edge label. For any node N in G there exists a path from BEGIN to N and a path from N to END. ■

Note that this definition of a CFG only allows nodes to have at most two possible successors. In general, it is possible for a CFG node to have multiple outgoing edges (such as with a 'case' statement). As will be seen in Section 4.3, the number of outgoing edges was restricted to make the definition compatible with NetGen's source language characteristics. A sample CFG is illustrated in Figure 3.1.

3.3 Data Flow Analysis

```
a)      PROGRAM smallest;

        VARa,b,c,smallest,i : INTEGER;
              sum,avg : REAL;

        BEGIN
   S1      sum := 0.0;
   S2      FOR i := 1 TO 10 DO
           BEGIN
   S3         readln(a,b,c);
   S4         IF (a <= b) AND (a <= c) THEN
   S5            smallest := a
              ELSE
   S6            IF (b <= a) AND (b <= c) THEN
   S7               smallest := b
                 ELSE
   S8               smallest := c;
   S9         sum := sum + smallest
           END

   S10     avg := sum / 10.0;
   S11     writeln('Average = ',avg)
        END.
```

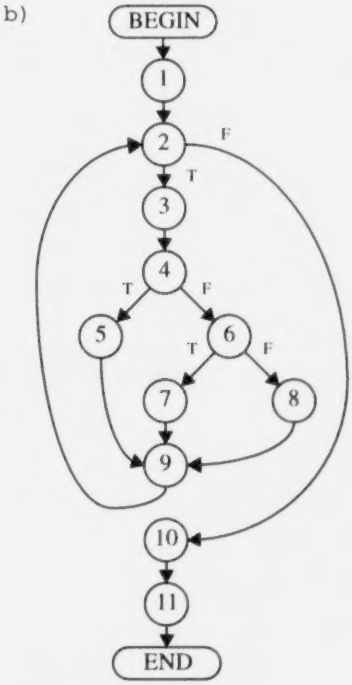


Figure 3.1 Example control flow graph. (a) a Pascal program, and (b) its control flow graph.

Data flow analysis attempts to determine how scalar variables are used in a program and is an important precursor to data dependency analysis. Data flow information collected from statements, basic blocks, and procedures can be conveniently represented using a mathematical set notation. An analysis of a program is completed by manipulating this information using basic set relations.

Data flow analysis has been studied extensively in the literature and many useful analysis techniques have been developed [AhSeUl86, Much81, ZiCh90]. Although much information can be collected from data flow analysis, only the information pertinent to NetGen's analysis will be discussed. The interested reader is referred to [ZiCh90, Burke90, AhSeUl86, and Much81] for a comprehensive survey of additional data flow analysis techniques.

Regardless of the level at which data flow analysis is performed, it is convenient to examine the use of scalar variables within that domain. Determining when a particular variable is defined and subsequently referenced is a fundamental necessity in data flow analysis. Informally, a definition of a variable X is a statement that assigns a value to X. A variable is used (or referenced) by a statement if its *r-value*[1] may be required [AhSeUl86].

All definitions and uses of scalar variables in statements can be represented in sets (commonly referred to as *def* and *use* sets, respectively). Statement sets can be further aggregated to produce *def* and *use* sets for basic blocks or entire procedures.

To acquire a more accurate representation of variable usage throughout a program, interprocedural data flow analysis is performed. This technique examines the flow of data across procedure boundaries and makes it possible to detect whether an entire procedure can be executed in parallel with another statement or procedure. Unfortunately, interprocedural analysis introduces a new problem, namely that of aliases to variables established by the parameters in procedure calls.

---

[1] an r-value is the value on the right side of an assignment.

There are three types of variables that can be used within a procedure: variables local to the procedure, variables declared outside a procedure but visible within the procedure, and variables passed as parameters to the procedure. Data flow analysis can proceed with little difficulty when considering the first two types of variables. However, procedure parameters can pose a special challenge.

Variables can be passed to a procedure in two ways: by value or by reference. When a variable is passed by value, a copy of the actual parameter is made and passed to the procedure. On the other hand, when a variable is passed by reference, the actual parameter itself is passed to the procedure. Parameter passing by value poses few problems in a data flow analysis. This is because a new, unique variable is declared at the start of the procedure and any changes to that variable modify the copy, not the original. However, when parameters are passed by reference it is now possible that two or more different variables refer to the same memory address. This situation is illustrated in the example program in Figure 3.2. As can be seen, variables A and X are aliases since they refer to the same memory location. Within procedure W, X modifies the contents of A. Variables B and Y are not aliases since Y is merely a copy of B at the start of the procedure and refers to a different memory location. All aliasing relationships must be identified before an accurate data flow analysis can be performed.

Aliases can also be introduced through the use of pointer variables. Analysis of code with pointer variables is generally a very difficult and involved task. The analysis can be simplified by restricting the source language to make pointers more "well-behaved" but the problem remains complex. For this reason, the initial implementation of NetGen does not allow pointer variables. Aliases introduced via pointers will not be discussed further, but the interested reader is referred to [AhSeUl86] and [ChWeZa90] for additional information.

3.4 Control Dependence

Control dependencies arise from the flow of control in a program. Whenever an imperative language is parallelized, control dependencies between statements must be observed.

a)

```
PROGRAM alias;

VAR A,B : integer;

PROCEDURE W(VAR X: integer, Y: integer)
BEGIN
   X := X * Y;
   Y := 0
END;

BEGIN
   A := 5;
   B := 10;
   WRITELN(‘A = ‘,A);
   WRITELN(‘B = ‘,B);

   W(A,B);

   WRITELN(‘A = ‘,A);
   WRITELN(‘B = ‘,B)
END.
```

b)

```
A = 5
B = 10
A = 50
B = 10
```

The program output demonstrates the aliasing relationship between variables A and X. It also shows how variables B and Y are not aliases.

Figure 3.2 Aliasing of variables via procedure parameters. (a) a sample program and (b) output from the program

As imperative languages rely on the propagation of side effects between statements to compute a result, the order of the statements is important. The definition of control dependence is based on the notion of post-dominance [Sarkar91a]:

Definition 3.2: Post-dominance

> Node Y *post-dominates* another node X ≠ Y iff every directed path from X to END in the CFG contains Y. ■

Post-dominance is used to determine nodes that have identical control conditions. In the above definition, if node Y post-dominates X, Y will always be executed whenever X is executed. Therefore X and Y have identical control conditions.

The definition of control dependence follows [FeOtWa87]:

Definition 3.3: Control Dependence

Let G be a CFG. Let X and Y be nodes in G. Y is *control dependent* on X iff:

1. there exists a directed path P from X to Y such that every node Z in P (excluding X and Y) is post-dominated by Y, and
2. X is not post-dominated by Y. ■

The first condition requires that for each node Z in path P, all paths from Z to END must pass through Y. The second condition requires that another path exist from node X to END in the CFG. Based on this definition, the control dependencies for each of the nodes in the sample program in Figure 3.1a are summarized in Table 3.1.

Node	Control Dependent On (Node, Edge)
1	--
2	--
3	2T
4	2T
5	4T, 2T
6	4F, 2T
7	6T, 4F, 2T
8	6F, 4F, 2T
9	2T
10	--
11	--

Table 3.1 - Control dependencies for the sample program in Figure 3.1a

Excessive control dependencies can limit the amount of parallelism that can be exploited in a serial program. For this reason, it is often desirable to remove all control dependencies from the subject program before parallelization begins. A common technique to eliminate control dependencies is *if-conversion* [AKPW83] and is discussed further in

Section 7.2.3.

The computation of control flow and control dependencies within a program can be done exactly with many imperative languages, including Pascal. This is because the targets of most branch statements can be determined at compile time. An assigned GOTO statement in Fortran, where the branch target is the value of a variable, is an exception. In this case, data dependence analysis would have to be used in lieu of control dependence analysis [Sarkar91a].

3.5 Data Dependence

A data dependency between two statements occurs when they share a variable access. Data dependencies arise from the flow of data within a program. Data dependency analysis is a complex and heavily researched field of study. Most of the recent research deals with the determination of data dependencies within loops, since much of the available parallelism in sequential programs can be found within loop structures. As the present NetGen project does not deal with parallelization of loop iterations, loop-carried dependency analysis will not be discussed.

The definition of data dependence follows from the literature [Sarkar91a, Almasi89]:

Definition 3.4: Data Dependence

> Let S1 and S2 be statements in a CFG. A *data dependence* (denoted S1 δ S2) exists between statements S1 and S2 with respect to variable X iff at least one of the following is true:
>
> 1. X is written by S1 and later read by S2 (flow dependence, denoted S1 δ^f S2), or
> 2. X is read by S1 and later written by S2 (anti-dependence, denoted S1 δ^a S2), or
> 3. X is written by S1 and later written by S2 (output dependence,

denoted S1 δ^o S2), or

4. X is read by S1 and later read by S2 (input dependence, denoted S1 δ^i S2). ■

The various types of data dependencies are illustrated in Figure 3.3. Of all the above dependencies, input dependencies have the least effect on parallelization. Usually, two independent reads to the same memory location (without any intervening writes in between) do not have an effect on the value stored there. Consequently, the same value will be retrieved regardless of the sequence in which the reads were performed. However, if the variable refers to a memory location whose contents are controlled by an external hardware device (such as an I/O processor or a timer) then the variable's value could change between successive reads. Variables of this type are called *volatile* and must be explicitly declared as such in the source language. [Sarkar91a] indicates that input dependencies can be ignored for correct parallelism but should be studied for efficient parallelism (e.g., reducing cache hot spots).

```
     .
     .
S1   y := 10
S2   x := y + z
S3   y := 12
S4   z := y
     .
     .
```

This code fragment yields the data dependencies:

S2 δ^f S1
S3 δ^a S2
S3 δ^o S1
S4 δ^a S2
S4 δ^i S2

Figure 3.3 Data dependencies between statements

Unlike control dependence analysis, data flow and dependence analyses are generally inexact. Consequently, whenever a result is in doubt, the solution favors a safe or conservative outcome.

3.6 I/O Dependencies

Another source of dependencies that can limit parallelism originate from the use of I/O statements. These typically appear as some form of READ or WRITE statement in a programming language. Often the order of I/O statements specified in a sequential program is important and should not be rearranged. For example, if a program were to display a textual message for the user, it would be unsatisfactory if the order of the message elements were changed. This could render the message unreadable. Consequently, some form of dependence analysis should be applied to I/O statements to prevent such problems from occurring.

The solution adopted by NetGen models I/O dependencies in a fashion similar to data dependencies. The I/O stream itself can be treated as a variable, and the various READ and WRITE statement dependencies modelled with a dependency on that stream.

3.7 Representing Dependency Information

Traditionally, some variation of the *dependence graph* [KKPLW81] is used in dependency analysis problems. A dependence graph is a directed graph that summarizes the dependencies between statements in a program. The nodes represent program components (such as statements) and edges between nodes indicate a dependency between the corresponding statements. Multiple dependency arcs between two nodes are permitted and a node may be dependent on itself. Conventional dependence graphs usually associate a *direction* with each dependency. This is beneficial if the exact type of data dependence between two statements must be known (e.g., is S1 δ^a S2?) or if a dependence analysis involving loops and arrays is to be performed. Control flow information is usually not preserved in such a representation.

An alternative structure for representing dependencies is the *program dependence graph (PDG)* [FeOtWa87]. Nodes represent statements or predicate expressions and edges represent both data and control dependencies.

> The program dependence graph explicitly represents *both* the essential data relationships, as present in the data dependence graph, and the essential control relationships, without the unnecessary sequencing present in the control flow graph [FeOtWa87].

This structure is useful in determining parallelism as only the necessary sequencing of statements is preserved.

As will be seen in the next chapter, the *dependency annotated net* is a third representation that is used in the NetGen project.

3.8 Conditions For Parallelism

Once all program dependencies have been determined, potential parallelism among statements can be identified. Parallelism between statements requires that they have no control, data, or I/O dependencies between them.

Applying the techniques outlined in this chapter will result in the identification of all possible statements that could be parallelized within a sequential program. The extent to which this information is used, however, depends on the remaining parallelization steps and on the resources of the target machine.

Chapter 4

A Program Partitioner For Net-Based Execution

4.1 Introduction

NetGen is an experimental program parallelization tool that produces a high-level net description of a program. Such a description is suitable for conversion into a Petri net representation. The focus of the parallelization is on sequential program partitioning, and several experimental parameters are available to guide this process. NetGen consists of several interconnected modules, as illustrated in Figure 4.1. Each module performs a particular analysis that is essential to the correct partitioning of the subject sequential program. The annotated net formation phase first determines the dependencies within the

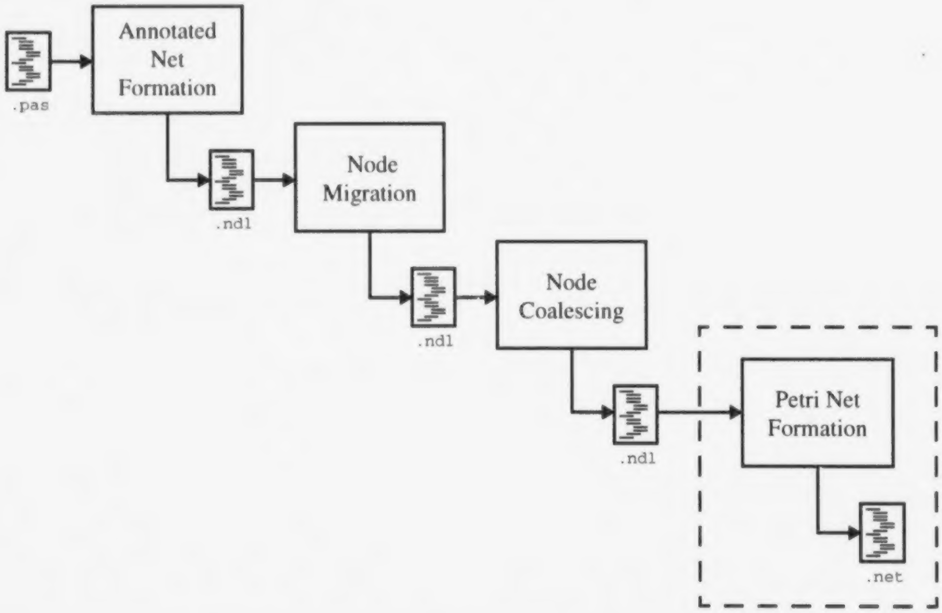

.pas
Annotated Net Formation
.ndl
Node Migration
.ndl
Node Coalescing
.ndl
Petri Net Formation
.net

Figure 4.1 NetGen's processing steps required to produce a net-based representation of a sequential program.

source program. Guided by the dependency analysis, the node migration stage arranges the code so that statements can be executed as soon as possible and permits parallelism to be expressed among these statements. Node coalescing can then be used to identify groups (or clusters) of statements that, depending on the criteria specified by the researcher, will be executed sequentially. The final, unimplemented stage of this project involves the translation of the partitioned program into a Petri net representation to be executed on a net-based architecture. The different modules communicate by passing a representation of the source program in a net description language. Each module adds information to the program description that is pertinent in subsequent analysis phases.

The modular approach to the parallelization process was chosen to clearly abstract the duties of each of the analysis phases. In addition, such an approach facilitates the creation of additional analysis steps that can be inserted into this process. For example, the current implementation deals only with the parallelization of Pascal programs. Additional language front-ends (such as Fortran or assembler) can be created to provide analysis capabilities to other programming languages.

This chapter discusses the first two stages in the net generation process. It will provide a suitable background for describing the node coalescing phase in the next chapter.

4.2 Target Architecture

As NetGen is an experimental tool that could be used in the search for architectures suitable for net-based computing, a specific target architecture cannot be identified. However, the architecture does exhibit general characteristics that the class of net-based machines will possess. In general, a tightly-coupled shared memory environment is assumed. There are a number of processing elements that communicate over a shared bus. A global memory, equally accessible to all processors, is available. Each processor has its own local memory that can be used for stack purposes. The system is driven by a task allocation unit, which essentially executes the Petri net description of the software and assigns appropriate tasks to processors. The general hardware configuration is illustrated in Figure 4.2.

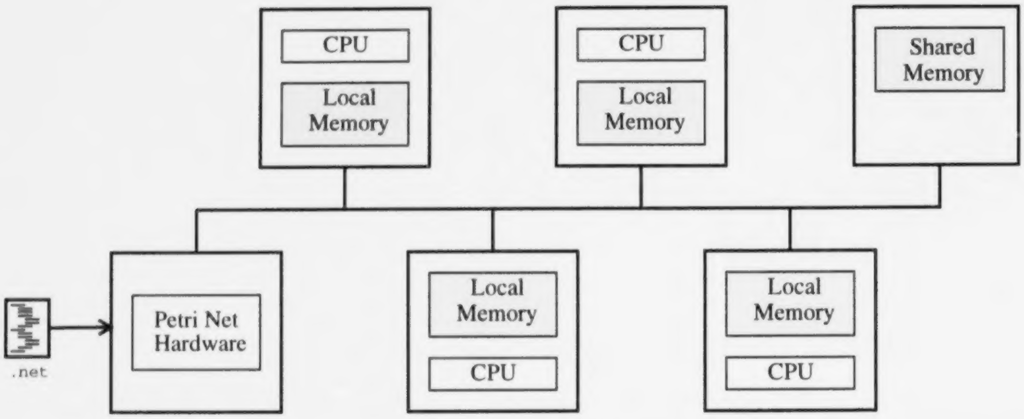


Figure 4.2 Execution environment for a net-based program

Each machine has a particular level of task granularity that it can best exploit. The partitioning of the software must therefore be done to accommodate the granularity requirements of the target architecture. The user can modify the coalescing parameters accordingly to influence the grouping of statements into subtasks.

4.3 Sequential Programming Language

For the NetGen project, the choice of an appropriate sequential programming language was based on both the features and programmability of the language, and on the ease of performing a program analysis. Consequently, a subset of the popular Pascal programming language was chosen to be the subject of the parallelization process. From the programmer's perspective, Pascal provides a rich set of programming constructs, operations, and data structures that make it suitable for the expression of many programming problems. From an analysis perspective, one of the most important characteristics of Pascal is that it is a block-structured language that allows the existence of multiple variable scopes. Since the dependency between statements with variables of different scopes is one of the parallelization properties that can be studied with NetGen, its support by the subject sequential language was imperative. Program flow and dependency analysis is also facilitated by the structure and context-free nature of the language.

A subset of standard Pascal [Cooper83] is sufficient to attain the objectives of this thesis.

The language was restricted primarily to ease the development of this initial analysis tool. However, some constructs were eliminated because they can be formed from other language elements. The major standard Pascal features that are not supported include the following:

- *goto statements.* In a language that provides loop constructs, **goto** statements are superfluous and can be eliminated [BoJa66].
- *pointer variables.* Although prevalent in many program designs, the effects of pointer variables are not a subject for investigation in this initial implementation.
- *case statements.* These can be emulated using cascaded **if** statements.
- *compound data types.* Compound data structures such as records and sets would add unnecessary complexity to this initial analysis tool.
- *file I/O.* Only the standard input and output streams are available.

Additional standard Pascal features can be added later and a study of the effects of their inclusion should prove insightful. Herein, the Pascal subset language used in the NetGen project shall be referred to simply as Pascal; additional qualifiers will be used if further clarification is necessary. A complete description of the standard Pascal subset grammar is provided in Appendix A.1.

4.3.1 Language Elements

There are several Pascal language features that are important to NetGen's sequential program analysis. These properties either have a direct analog in the dependency annotated net or are essential in understanding its construction. This section attempts to clarify some of the concepts and terminology that will be used when describing how a net is generated from a Pascal program.

4.3.1.1 Statements

There are two types of statements available in the Pascal language: simple and structured statements. Variable assignments and procedure calls are *simple statements* whereas compound statements, conditionals, and loops are considered *structured statements*. The distinction between simple and structured statements becomes important when nodes are created for the annotated net.

4.3.1.2 Blocks, Regions, and Scope

Pascal is a *block-structured* language. A Pascal program is composed of a number of sections (called *blocks*) that can contain variable declarations, constant definitions, procedure declarations, and statements. The program itself is contained in the program block while procedures and functions reside in separate procedure and function blocks, respectively. One of the useful properties of blocks is that they can be nested, giving rise to the notion of regions and scope.

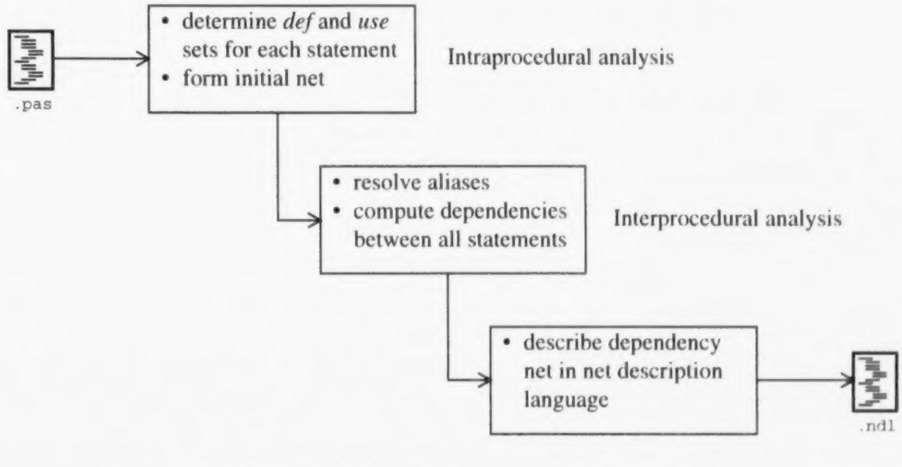
"A block (and any blocks it contains) constitutes the *region* in which...[an] identifier can retain its original meaning." [Cooper83] An identifier declared within a block is accessible to all procedures and functions declared within that block. Subsequent blocks declared within those subroutines will also recognize the identifier. Regions are important as they define the largest possible program domain in which a particular identifier can be used. An identifier's *scope* is a closely related concept—the terms are often confused. The scope of an identifier is the actual program area in which the identifier is visible. It may be either the same size or smaller than its region. Scopes arise from the nesting of blocks and limit the region of an identifier only when a similarly named identifier is declared within one of those nested blocks.

A *global identifier* is one that is declared in the program block—its region is the entire program. A *local identifier* is declared within a procedure or function block and is visible only to the blocks nested within it. NetGen also uses the terms *relatively local* and *relatively global* to refer to the scope of an identifier. Both are very subjective terms and are dependent on the block in which they are applied. A relatively global identifier is one whose scope completely encompasses a block in which it is used. In other words, it is declared outside the current block. A relatively local identifier is one whose scope has limited the visibility of an identically named identifier in an outer block.

4.4 Annotated Net Generation

The first step in the parallelization process is the construction of a dependency annotated net for the sequential program. Basically, an annotated net is a description of the dependencies between statements in the sequential program. This section describes `makenet`, the tool used to accomplish this analysis. As this is the only phase that deals directly with the original sequential program, this step must be performed first. Subsequent stages can manipulate the description of the dependency net that is produced by this module.

`makenet` employs the processing steps depicted in Figure 4.3 to produce a dependency annotated net. The first step parses the sequential Pascal program to be parallelized. An intraprocedural analysis is performed on each subroutine to determine the *def* and *use* sets for each statement. The statements are initially mapped into a net structure without any dependency information. The second analysis phase performs an interprocedural analysis of the nets produced in the previous step. Aliasing of identifiers is resolved and the nets are annotated with dependency information by computing the dependency relationships



.pas
• determine *def* and *use* sets for each statement
• form initial net
Intraprocedural analysis
• resolve aliases
• compute dependencies between all statements
Interprocedural analysis
• describe dependency net in net description language
.ndl

Figure 4.3 `makenet` processing steps between statements. Finally, the net information is transcribed into a file for the next parallelization module. Each processing step will be described in detail in subsequent sections.

4.4.1 Dependency Annotated Net

NetGen performs most of its analysis on an intermediate dependency net representation of the source program. The net representation structure used is the *dependency annotated net*, which is similar to the conventional dependency graph described in the previous chapter. A dependency annotated net (or simply dependency net) is an undirected graph that summarizes both the essential dependencies between statements and their relative execution order. A sample dependency net is illustrated in Figure 4.4c. The net is augmented with a unique entry node NET_INITIALIZE and a unique exit node NET_TERMINATE. All other nodes in the net represent either a Pascal simple statement, a loop predicate, or a condition predicate[1]. Table 4.1 lists all possible node types. Dependencies are represented as directionless edges between nodes and at most one dependency edge can exist between any two nodes. This has two implications. First, there is no concept of dependence direction associated with an edge. Since only a scalar analysis is performed and parallelization across loop iterations is not investigated, dependence directions are not needed. Second, the number of dependencies between statements is not important for determining execution order or possible parallelism. The mere existence of a single

Node Type	**Description**
NET_INITIALIZE	Entry point to the net. It is always the first node executed.
NET_TERMINATE	Exit point of the net. It is always the last node executed.
ASSIGNMENT	Assignment statement
PROCEDURE_CALL	Procedure call
IF_PREDICATE	Predicate for a condition statement
LOOP_HEADER	Loop header for a FOR or WHILE loop
IO_PROCEDURE	Standard I/O procedure (READ or WRITE)

Table 4.1 - Types of nodes available in the dependency annotated net

---

[1] Herein, references to nodes and statements will be considered equivalent.

edge is sufficient for analysis purposes. Although the number of dependencies between statements can be important during the node coalescing phase, that information can be easily computed when required.

This net representation does not describe a program in a Petri net notation. Instead, a format that facilitates the identification of dependencies between statements is employed. However, the dependency annotated net contains sufficient information to generate a Petri net description of the source program following NetGen's analysis. This intermediate net structure should not be confused with the net description language used as the communication medium between modules. Although similar, the net description language merely provides a textual representation of the intermediate net structure and will be discussed in Section 5.7.

A separate dependency net is produced for each block in the source program. The dependency information used to construct the net is inferred from a static analysis of each block. Every dependency net created has the characteristics described in this section.

The execution order of statements is determined by the dependencies between them. In an imperative language, the dependencies between statements are transitive and often many are redundant when specifying the correct execution order. Consider the program fragment in Figure 4.4a that contains both data and control dependencies. A conventional dependency graph would have edges similar to that shown in Figure 4.4b. As can be seen, node 4 has a dependency with nodes 1, 2, and 3. Since node 4 has a dependency with node 3, node 3 has a dependency with node 2, and node 4 must be executed after both nodes 2 and 3, then a dependency edge is not necessary from node 4 to node 2. All the dependency edges needed to specify the correct execution order are illustrated in Figure 4.4c. Hence, although there may in fact be a dependency between two statements, a dependency edge need not be created if they are already indirectly connected via other dependency edges. This is the approach taken by NetGen when constructing a dependency annotated net.

```
a)          .
            .
            .
S1   c := a * 4;
S2   a := b * c * 10;
S3   if (a < 256) then
S4      b := c
     else
S5      b := c * 2;
            .
            .
            .
```

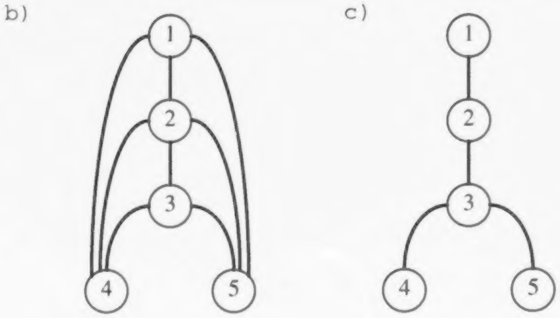


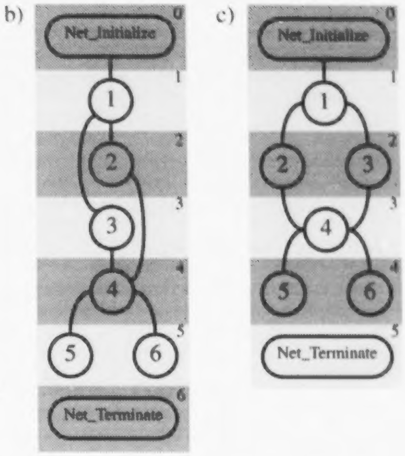
Figure 4.4 Demonstration of unnecessary dependencies. (a) a program fragment, (b) all dependency relationships, and (c) its minimal dependency relationships

The correct ordering of statements in the dependency net is specified not only by the dependencies between them but by mapping nodes into *execution steps*. An execution step specifies all the statements that can be executed together at a particular "step" in the program execution without any real time considerations. A net is composed of several execution steps that collectively describe the relative sequence in which statements must be performed. An execution step only contains statements that have no dependencies with one another. However, they may have dependencies with statements in other execution steps. Statements that share an execution step can potentially be executed in parallel (i.e., they have no dependencies between them and can therefore be executed at the same time).

Figure 4.5 illustrates a sequential program's mapping into execution steps. Figure 4.5b shows the minimum set of dependencies for the program that are computed following the net annotation phase. It is apparent that nodes 2 and 3 do not have any dependencies with one another and could potentially be executed in parallel. Consequently, nodes 2 and 3 can share an execution step, as depicted in Figure 4.5c. The purpose of the node migration phase (discussed in Section 4.5) is to determine which nodes can share an execution step.

Execution steps are numbered consecutively beginning at zero. In general, the next statement to be executed following the current statement is found in a subsequent (or numeri-

a)
```
        PROGRAM triangle;

        VAR a,b,c,d,e: integer;

        BEGIN

S1         readln (a,b,c);
S2         d := a*a + b*b;
S3         e := c*c;

S4         if (d = e) then
S5            writeln('Right triangle')
           else
S6            writeln('Not a right triangle')

        END.
```



Regions shaded ▨ and ░ represent execution steps. The execution step number is indicated in the upper-right corner of each execution step.

Figure 4.5 The mapping of a program into execution steps. (a) a simple program, (b) its dependency annotated net, and (c) its dependency net following node migration

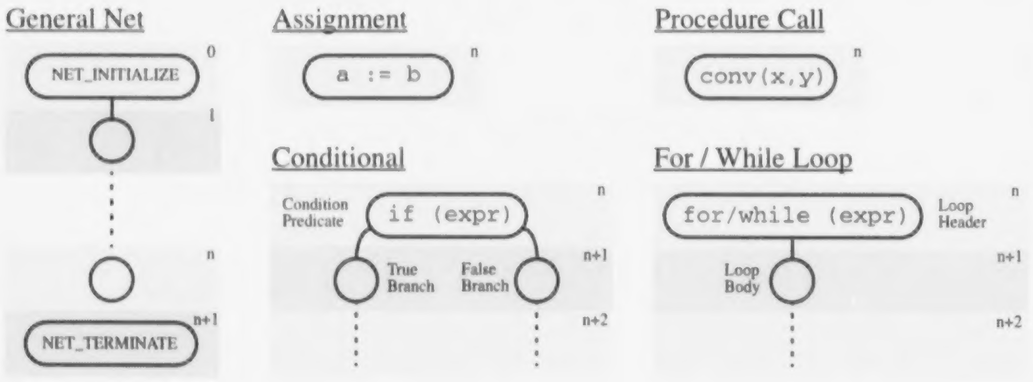
cally larger) execution step. The exception to this rule is with loop back edges which appear at the end of each loop body toward the loop header. Figure 4.6 illustrates how the various statements from the Pascal source are mapped into execution steps.

The NET_INITIALIZE node resides alone in the first execution step of the net while the NET_TERMINATE node is always placed in the last possible execution step. The former represents the initialization of any execution context (such as the creation of a stack) that must be performed *before* any statements in the net are executed. The latter node is the complement of the NET_INITIALIZE node: it represents the deallocation and restoration of the execution context once the net has completed execution. Consequently, no other nodes may appear in the same execution step as these nodes.

All nodes have an implicit control dependency on the NET_INITIALIZE node. This is done partly to ensure that all nodes are executed after the subroutine has had a chance to properly initialize its execution context before any statements are executed. It is also correct to assume a control dependency with the entry node because it can be interpreted as a boolean condition that determines whether or not the subroutine will be executed. A similar approach is taken by [FeOtWa87] in their program dependency graph, but they insert an explicit ENTRY node into the graph to handle the boolean condition. Note that this control dependency edge is treated like all other dependencies: if an indirect path of dependencies exists to the NET_INITIALIZE node then an explicit edge is not required.

The NET_TERMINATE node does not have any dependencies with any other node. Although this is usually an indication for parallelism, due to the special properties of the NET_TERMINATE node it is always placed last in a net.

The statements in the branches of a conditional must be executed after the condition predicate is evaluated. The outcome of the evaluation determines which branch will be taken. Consequently, every node in the branches is control dependent on the predicate node and hence a dependency edge is required. Note that the True and False branches may be different lengths (i.e., they contain a different number of statements). Both branches still begin in the next execution step from the predicate node. However, the next statement fol-


General Net
NET_INITIALIZE
0
1
n
NET_TERMINATE
n+1
Assignment
a := b
n
Procedure Call
conv(x,y)
n
Conditional
Condition Predicate
if (expr)
n
True Branch
False Branch
n+1
n+2
For / While Loop
for/while (expr)
Loop Header
n
Loop Body
n+1
n+2

- each branch of the conditional statement is control dependent on the condition predicate node
- all statements within the body of a loop are control dependent on the loop header node

Figure 4.6 Mapping Pascal statements into a dependency net

lowing the condition statement begins in the next execution step after the longest branch path. A similar situation exists with loops. The loop header node must be executed before any of the loop body statements. This constraint appears as a control dependence between the loop header and every statement in the body of the loop.

4.4.1.1 Dependency Edges

The dependencies within an annotated net can be classified into two types: hard or soft dependencies. A *hard dependency* is a single, directionless dependence edge between two nodes that represents all control, data, and I/O dependencies between them. It eliminates the need to have several dependence arcs between nodes. Any information about the number or types of dependencies that the hard dependence edge is representing is not retained. Only the existence of a dependence edge is sufficient for most analyses. Hard dependencies play an important role in influencing statement sequencing and identifying potential parallelism.

Soft dependencies, on the other hand, exist between statements that do not lie on the same control path. In other words, a soft dependency appears between statements that can never be executed at the same time. This type of dependency is found between statements that are in different branches of the same conditional statement. Although both branches can be executed, only one branch will ever be executed at a time in a particular execution environment (e.g., a procedure). A soft dependence edge is also directionless but it does not affect the ordering of instructions. Soft dependencies do allow nodes that are in different control paths to share an execution step, a privilege usually reserved for nodes that could be executed in parallel. Since both nodes will not be executed at the same time, they can be thought of as candidates for potential parallelism. Soft dependencies are also used to determine the next statement to be executed following the node in the current execution step. The next statement must be in the same control path, and therefore the choice of the next node to execute can be made by considering only those nodes without a soft dependency with the current node.

Figure 4.7 illustrates both hard and soft dependencies in the dependency net for the example program in Figure 3.1. For the remainder of this thesis, hard dependence edges between nodes will be represented as solid lines, whereas soft dependence edges will be depicted using dashed lines.

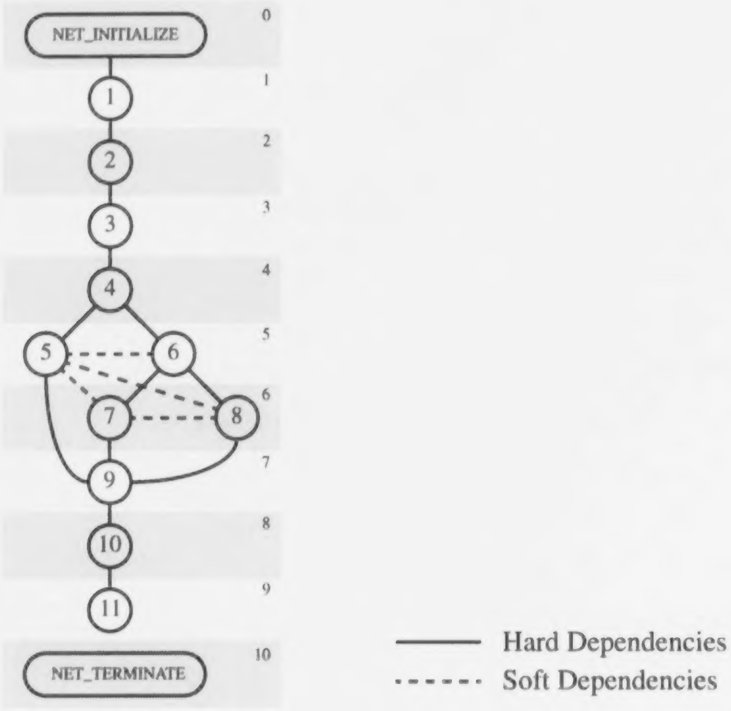


Figure 4.7 Hard and soft dependencies in the annotated net for the sample program in Figure 3.1.

4.4.1.2 Dependency Regions

Besides the explicit hard dependencies and execution steps, grouping nodes into dependency regions can also affect the statement parallelism determination. A *dependency region* is a collection of nodes that can only be parallelized amongst themselves and where each collection is delimited by structured constructs of the language. Each block forms a dependency region since parallelism is exploited within the confines of a single procedure. Every loop also begins a new dependency region since loops have special sequencing and parallelism requirements. Although parallelism across loop iterations is not explored in NetGen's analysis, parallelism opportunities within a single iteration can be examined. As a result, a special structure is needed to isolate a single loop iteration from the other non-loop statements. Figure 4.8 depicts the various manifestations of dependency regions.

In general, nodes outside a dependency region cannot be executed in parallel with nodes inside the region. This is certainly not true for blocks—execution of statements in different blocks in parallel is simply not permitted in NetGen. However, under some circumstances nodes outside a loop can be executed in parallel with the loop as a whole. The problem with allowing loop and non-loop statements to be considered for parallelism is clear: node coalescing that is performed at a later phase might group a non-loop node into the loop. This is unacceptable since the non-loop node would then be executed with each loop iteration.

Dependency regions can be nested. Each region within a block is assigned a unique integer number for identification purposes. Every new block is assigned zero as its dependence region identification. Any region created within a block dependence region is associated with a loop. At present, no other structures in NetGen require a separate dependency region for analysis.

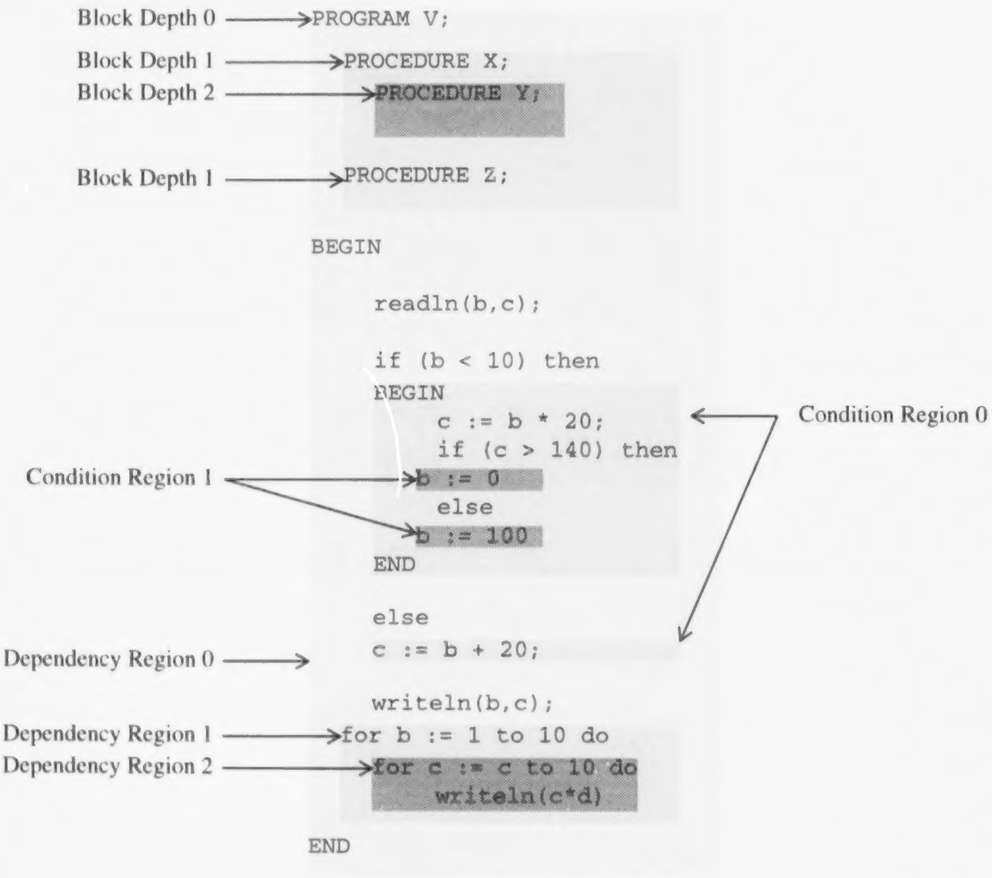
Dependency regions have only one entry point. The first node is referred to as the *region header node*. For blocks the NET_INITIALIZE node is the region header, whereas for loops the loop header node is the start of the dependency region. A dependency region nested within another region can be treated simply as another node in the net. Dependencies can be computed with it and it can be moved for parallelization purposes. Every dependency region maintains a set of all external nodes (i.e., outside the dependency region) to which it has a dependency. These are called *efferent region dependencies* since they are the set of all dependent nodes that lie outside the dependency region. This information is compiled by ignoring the fact that the statements are in separate dependency regions and applying the traditional analysis techniques. Thus, a statement outside a dependency region can be executed in parallel with the region if it has no dependencies with any statement in the region. If this is the case then the external statement could be placed in the same execution step as the region header node.

4.4.1.3 Condition Regions

Every condition statement is associated with a unique *condition region*. A condition region associates the branches of a condition statement to a particular condition predicate. These regions do not affect program parallelization. All branches of the same condition statement belong to the same condition region. These regions can be nested and each is assigned a unique condition region number. Figure 4.8 illustrates how condition regions are assigned.

4.4.1.4 Block Depth

Every block that is created in a Pascal program is nested a certain number of levels within


Block Depth 0
Block Depth 1
Block Depth 2
Block Depth 1
Condition Region 0
Condition Region 1
Dependency Region 0
Dependency Region 1
Dependency Region 2

```
PROGRAM V;
   PROCEDURE X;
      PROCEDURE Y;

   PROCEDURE Z;

BEGIN

      readln(b,c);

      if (b < 10) then
      BEGIN
          c := b * 20;
          if (c > 140) then
        b := 0
          else
        b := 100
      END

      else
      c := b + 20;

      writeln(b,c);
   for b := 1 to 10 do
      for c := c to 10 do
          writeln(c*d)

END
```

Figure 4.8 Block depth, dependency regions, and condition regions

the program block. The level at which a block is nested with respect to the program block is called the *block depth*. The program block is at a block depth of zero. More than one block may be at a particular block depth. Figure 4.8 illustrates how block depths are computed for various nested procedures.

4.4.2 Intraprocedural Program Analysis

The translation of a sequential program into a net-based representation begins with an intraprocedural analysis of the Pascal source. This analysis is used to infer important information from each subroutine including its control structure, the declaration of nested subroutines and variables, and the usage of such variables in each statement. The result of this initial analysis phase is the production of a rudimentary net description for every Pascal block. The structures produced are not yet dependency nets since they do not contain any information about the dependency relationships between statements.

4.4.2.1 Parsing The Sequential Program

The first step in the intraprocedural analysis is the bottom-up parse of the Pascal source. Pascal is a simple language to parse as it has an LL(1) type of grammar. The language is structured such that only one pass of the source is required to extract all necessary information. Subsequent analysis stages deal only with the information learned from the initial parse and hence the source program is not examined again.

The analysis proceeds by constructing a separate net for every Pascal block encountered. A NET_INITIALIZE node is created and added to the first execution step of the new net structure. The statements within a block will eventually form the nodes of that net. Each statement is analyzed individually. The line number within the source file where a particular statement was encountered is stored since it is required in the final Petri net generation stage.

4.4.2.2 Declarations

Before the statement part of a block there are several sections that allow the declaration of constants, variables, and other blocks. All declarations involve the use of an identifier that assumes a unique meaning in the current block. For constants and variables, the identifier refers to the name of the constant or variable; for other block declarations, the identifier refers to the name of the procedure or function.

For constant declarations only the identifier name is retained. Its corresponding value is not important in any analysis performed by NetGen and is extraneous. Although procedure and function declarations begin new blocks, their names are stored locally to associate them with the current block.

Variable declarations are treated somewhat differently as they are the principal elements of a data dependency analysis. Each variable declaration is given a unique declaration number. Declaration numbers begin at zero and apply throughout the entire source program. This is convenient during data flow and dependence analyses since some method is required to distinguish the variables involved. The declaration number is stored along with the variable identifier in the current block. Array variables are treated as scalar variables and are stored as such. Any access to a particular element of the array is treated as an access to the entire array. Variable data types are not retained, as the semantic properties of variables are not an issue in dependency analysis.

A section for type definitions also precedes the statement part of a block. It allows the programmer to rename types to improve the readability of the code. Like the data types associated with variable declarations, the information contained within this section is superfluous and is ignored by NetGen.

4.4.2.3 Dependency Net Nodes

The statement part of a block is the source of all nodes in a net. The statements are analyzed in the order in which they appear in the source program. When a conditional statement is encountered, the True branch is analyzed first, followed by the False branch. When a loop statement is detected, its loop body is analyzed immediately after the loop header. The definition of a node and the proper procedure for inserting a node into the net has already been specified in Section 4.4.1.

Every node is assigned a unique node number. Node numbers begin at zero and apply only to the current net. These numbers are assigned based on the order in which statements are encountered during the program parse. A convenient property of node number assignments is that a statement executed prior to the current statement will have a relatively smaller node number.

When a new node is encountered there are several pieces of supplementary information that must be retained for subsequent analysis stages. This information is summarized in Table 4.2. Some of this data cannot be supplied during the intraprocedural analysis phase and must be inserted later following an interprocedural analysis.

Symbol	**Description**
NN	Unique node number
Ntype	Type of node
Src	Statement text from Pascal source
LN	Line number in Pascal source file where statement is located
Estep	Execution step in which this node is a member
HD	Hard dependency edge set (refers to nodes)
SD	Soft dependency edge set (refers to nodes)
DD	Data dependence set (refers to nodes)
CD	Control dependence set (refers to nodes and edges)
IO	I/O dependence set (refers to nodes)
Eff	Efferent region dependencies (refers to nodes)
def	*def* set (refers to variable declarations)
use	*use* set (refers to variable declarations)
DR	Dependency region
RHN	Region header node
CR	Condition region

Table 4.2 - Data retained by nodes in a dependency annotated net

As each statement is encountered, the *def* and *use* sets are computed. The variables used or defined by the statement are stored via their declaration numbers in the appropriate data flow set. The *def* set represents all the variables that have been modified by the current statement. Only assignment statements, FOR loop headers, and READ procedures can modify a variable in the Pascal subset recognized by NetGen. The *use* set contains all variables that are read by the current statement. As will be seen in Section 4.4.2.5, the data flow sets for a procedure call statement are further supplemented with all the variables defined and used within that procedure.

All nodes are associated with the current dependency region and/or condition region under analysis. Conditional statements begin a new condition region and loop statements begin a new dependency region. Both types of regions remain “open” until they are closed by the termination of the corresponding loop or condition statement. As a result, other regions may be nested within open regions. The condition predicate node is not part of the new condition region that it creates; it is associated with the previous open condition region. New loop header nodes, however, are associated with the new dependency region.

4.4.2.4 Control Dependence

The control dependence between nodes can be computed during the intraprocedural analysis phase. Unlike data dependency analysis, control dependence information can be determined at this stage because control flow within one procedure is not influenced by the side effects of another.

An explicit control flow graph is not generated for each block. Although the notion of a CFG is essential for *defining* control dependence it is not required because control flow can be directly inferred from parsing the source program. Similarly, control dependence can also be determined from the parse. From the definition of control dependence (Definition 3.3), it follows that nodes can only be control dependent on nodes that have more than a single branch target. The only statements that satisfy this requirement in the Pascal language are conditional statements and loop headers. Consequently, any node is immediately control dependent on the last conditional or loop header encountered. The branch (True or False) from the control dependent node is also required to complete the control dependence set. Since the sequential program is parsed such that its statements are encountered in the order in which they are executed, determining control dependence via this method is appropriate.

If a control dependence between two statements is identified, a hard dependency edge between them is not inserted until after the interprocedural analysis. Only the control dependence set is retained. Since a dependency edge will not be added if another (indirect) dependence path already exists between two nodes, it would be premature to add an edge before the data dependence analysis is complete.

4.4.2.5 Analysis Of Multiple Blocks

When a procedure or function block is encountered during parsing, special consideration is required upon forming a new dependency net. Unlike the program block, parameters can be passed to procedure and function blocks. Consequently, the formal parameters and their parameter-passing method (by reference or by value) must be retained. Such parameters are the source of aliasing relationships between variables in different blocks.

Each formal parameter is added to the *def* set of the NET_INITIALIZE node for the new block. These parameters can be thought of as being defined upon entry to the block. When the block is completed, the NET_TERMINATE node is inserted into the dependency net. The union of all *def* sets within the net is computed and stored in a special *block def* set. A *block use* set is similarly determined from the *use* sets. This information clearly represents all the variables that have been defined or used within a procedure or function.

These sets are particularly useful at procedure or function call sites, where additional information is computed for data dependence analysis purposes. The variables contained within the *block def* and *block use* sets of a subroutine are added to the *def* and *use* sets of the call statement. This is done to provide additional information for parallelization purposes: a subroutine call can be executed in parallel with another statement if the subroutine code has no dependencies with the other statement.

Subroutine call sites are also where aliasing between actual and formal parameter variables is determined. During this intraprocedural analysis phase, possible aliases between variables are merely noted; the determination of variable equivalence classes is performed during the subsequent interprocedural analysis. Each actual parameter is matched to the corresponding formal parameter of the subroutine. If the parameter is passed by reference then an alias is indicated between the formal and actual parameter. Conversely, if the parameter is passed by value then an alias relationship is not established (since only a copy of the variable is passed).

4.4.3 Interprocedural Analysis

After a rudimentary net has been formed for each block in the source program, the interaction among these blocks must be determined. To this end, aliases across block boundaries are identified and the dependencies between all nodes in a net are computed. At this point, the parse of the Pascal source has been completed. Herein, only the internal structures that contain information learned from the parse are consulted; the source is no longer examined.

4.4.3.1 Alias Resolution

In order for the dependencies between statements to be accurately determined, all variable aliases must be identified. With the Pascal language subset used in the NetGen project, the only source of variable aliases are procedure calls. More specifically, if a parameter is passed by reference a possible alias exists between the formal parameter variable and the actual parameter variable. Consequently, the search for aliases requires an interprocedural analysis.

NetGen uses a simple algorithm for resolving aliases. Its basic premise is that if any two nodes could ever possibly be aliases of one another then they are always made so. Although simple, this approach is rather restrictive and can lead to the assumption of "false" aliases. A more complex analysis is required to eliminate such false alias relationships. The approach taken, however, is clearly a safe technique because no aliases are missed.

The alias determination algorithm has been adapted from Algorithm 10.12 in [AhSeUl86]. Instead of immediately updating the *def* and *use* sets of every statement that employs an aliased variable, the aliasing relationships are stored and are applied only when each node is visited during the dependency calculations. The algorithm for resolving variable aliases is presented in Appendix B.1. The result of applying the algorithm to the declarations determined from the intraprocedural program analysis is the mapping of aliased variables into equivalence classes. A variable can be a member in only one equivalence class. Accessing any variable within a particular equivalence class is treated as an access to all variables within that set.

The algorithm in Appendix B.1 resolves both the reflexive and transitive types of aliasing situations. For example, suppose X, Y, and Z are variables. If X and Y are aliases then it is desired that X and Y be put into the same equivalence class (i.e., $X \equiv Y$ and $Y \equiv X$). In addition, if X is an alias of Y and Y is an alias of Z then X should be made an alias of both Y and Z (i.e., $X \equiv Z$ whenever $X \equiv Y$ and $Y \equiv Z$). The simple algorithm presented is adept at handling both cases.

4.4.3.2 Dependency Computation

Once all the variable aliases have been resolved, the dependencies between nodes can be safely computed. As indicated earlier, given the role of execution steps and the transitive nature of the dependencies, a dependence edge will not be inserted between two nodes if an indirect dependency already exists between them. The addition of the new dependency edge would be redundant. Data, control, and I/O dependency conditions are checked between nodes within the same control flow, and soft dependencies are inserted between nodes in different control flows.

The algorithm for computing this minimal set of dependencies is ComputeDependencies and is described in Appendix B.2. As can be seen on line 1, the algorithm traverses every net produced. Within a net, the nodes are analyzed in the order in which they were added to the net during the intraprocedural analysis phase. When a node is encountered, every node executed prior to that node in the net is checked for a dependency. If a dependency is found, and an indirect path of dependencies does not already exist between them, then a hard dependency edge is inserted.

Line 8 checks whether or not two nodes have the same control dependence set. If they do not then they cannot be executed in the same control flow and a soft dependency edge is placed between them. A hard dependency edge cannot be placed between nodes with a soft dependency. Line 11 checks for a control dependence, while lines 13, 16, and 19 check for flow, anti, and output data dependencies, respectively. Dependence on an I/O resource is examined in line 22.

If a dependency exists between nodes then a hard dependence edge may be inserted between them. Nodes that lie in different dependency regions (line 26) must have the efferent region dependencies updated. Finally, line 29 checks whether an indirect path of dependencies already exists between the current node and the previously inserted node. If such a path exists, then a new edge is not required. Otherwise, a hard dependence edge is inserted in the net between the two nodes (lines 30-31) and the appropriate dependency sets are updated accordingly (lines 32-37).

PathExists is an important support algorithm for ComputeDependencies. It essentially determines whether or not two nodes are connected via some path of hard dependencies in a dependency net. The algorithm itself is straightforward and is similar to other connected component determination algorithms. Given two nodes *n* and *m*, where *n*.Estep > *m*.Estep, the algorithm starts at node *n* and searches for node *m* following existing hard dependency edges in the net. The conditional on line 10 steers the search to nodes that have been analyzed previously. If the search does not reveal node *m* then a dependency edge must be inserted since an indirect path cannot be found. Conversely, if node *m* is reached then a dependency already exists between them.

4.4.4 makenet Invocation Syntax

The analysis tool that generates a dependency annotated net is a stand-alone executable called `makenet`. It is invoked from the command-line with the syntax:

`makenet` <Pascal source filename> <output annotated net filename>

Although the *syntax* of the sequential Pascal program is verified by `makenet`, this module does not perform any *semantic* checking of the source code. It is assumed that the program to be parallelized has already been tested and executed in a sequential environment and can be compiled successfully. However, any errors or anomalies that may be encountered during the analysis process will result in the immediate termination of the net generation procedure. An informative diagnostic message is generated to assist the user in isolating the problem. Upon successful completion of the program analysis, the resulting annotated net will be described using the net description language and stored in the specified output file.

4.5 Node Migration

The next phase in NetGen's program parallelization process is *node migration*. It is this stage of the analysis where parallelism among nodes is revealed. Nodes are moved to the earliest possible execution step in which they can begin executing, possibly sharing that execution step with other nodes. The existence of multiple nodes in the same execution step is an indication of potential parallelism among those nodes. The movement (or migration) of nodes is restricted by the existence of hard dependency edges within the dependency net. These dependencies must be strictly obeyed. Failure to adhere to the dependency constraints between nodes established in the net will result in an incorrect program partitioning. This section describes `nodemig`, the module responsible for performing the node migration.

`nodemig` manipulates the dependency annotated nets produced by `makenet`. The algorithm used to migrate nodes is described in Appendix B.3. The three main components of this algorithm are MigrateNodes, AddNode, and DependencyExists. MigrateNodes is responsible traversing each of the annotated nets that have been produced (line 1). A new, empty net is created for each annotated net (line 2). The execution steps of the current net are then traversed in increasing order and each of their constituent nodes are added to the new net via the migration procedure (line 5).

The AddNode algorithm performs the node migration procedure. Basically, it places nodes in their earliest possible execution step governed by the dependency edges in the new net. Only the dependency edges to nodes that have been added previously are important when adding a new node. Since the goal is to place the node in its earliest execution step, this implies that only nodes that reside in a previously created execution step should be considered in the analysis. Starting with the execution step in which the candidate node originally belongs, each existing execution step in the new net is traversed in decreasing order and checked as a possible receptor for the candidate node. If any dependencies exist with the receptor execution step, the migration cannot proceed and the node is placed in the last execution step in which it did not have any dependencies.

In algorithm AddNode, the lastSafeEstep on line 1 refers to the last known execution step in which the current node can be placed without any conflicts. Each execution step from the current node to the start of the net is checked as a possible receptor for the node (line 2). A candidate node cannot have a dependency with any node in a possible receptor execution step (line 4). The existence of such a dependency would violate the definition of parallelism between nodes and the migration is therefore not permitted. A node is eventually placed in the execution step immediately following an execution step in which it had at least one hard dependency. Its new execution step will occur no later than the original execution step.

The dependency regions of the candidate node and the nodes within the possible receptor execution step determine the source of dependencies that need to be checked in order to allow the migration. A candidate node must check every node in the potential execution step. There are three possible node migration scenarios. First, the candidate node and a node within the receptor execution step can lie in the same dependency region. In this case, the existence of a hard dependency between the nodes is sufficient to deny access. Consider the dependency net in Figure 4.9. If node w is to be migrated into execution step n then only the hard dependencies between nodes v and w need to be checked.

Second, the candidate node has a larger dependency region number than a node within the

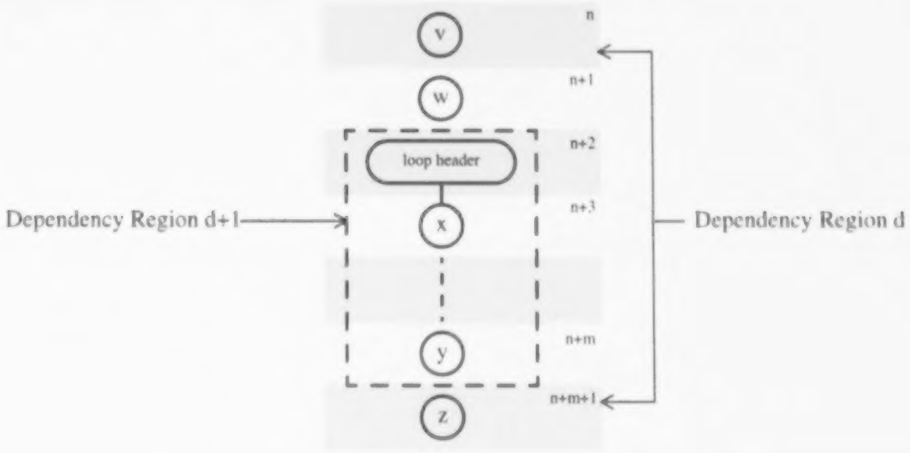


Figure 4.9 Determination of dependencies between nodes in different dependency regions during node migration

possible receptor execution step. This indicates that the candidate node lies within a loop structure and is being compared to a node outside the loop. Since nodes within the loop body must be kept together, any migration of the node must be acceptable to all nodes within the loop. Consequently, the node in the possible receptor execution step is checked for membership within the efferent region dependencies set of the loop. If such a dependency exists then migration is not allowed. Note that it is possible that the candidate node and the node outside the loop did not share a hard dependency. However, the existence of a dependency from some other node within the loop is sufficient to deny migration. Consider the illustration in Figure 4.9. This second scenario occurs if, for example, the loop header node were to attempt migration into execution step *n+1*. Since different execution steps are involved, there must be no dependencies between node *w* and any node in the loop (this includes the loop header and nodes *x* through *y*) for the migration to proceed.

The final scenario is similar to the previous case. In this scenario, the candidate node has a smaller dependency region number than a node within the possible receptor execution step. This indicates that the candidate node is being compared to a node within a completed loop structure. A node cannot be executed in parallel with a loop unless it has no dependencies with any of the statements within the loop. Consequently, the candidate node is checked for membership within the efferent region dependencies set of the loop. If such a dependency exists then the migration is not allowed. Again consider Figure 4.9. This scenario occurs if node *z* were to attempt migration into execution step *n+m*. If no statement within the loop has any dependencies with node *z* then the migration is allowed.

The dependencies between nodes are checked with the DependencyExists support algorithm. The first scenario is tested in line 1, the second on line 4, and the third on lines 7-8.

The migration of a node is illustrated in Figure 4.10. Node *x* begins in execution step four (a). Since it does not have any dependencies with any node in execution step three it can migrate into that execution step (b). However, it still does not have any dependencies with a node in the previous execution step. Consequently, node *x* can migrate once more into execution step two (c).

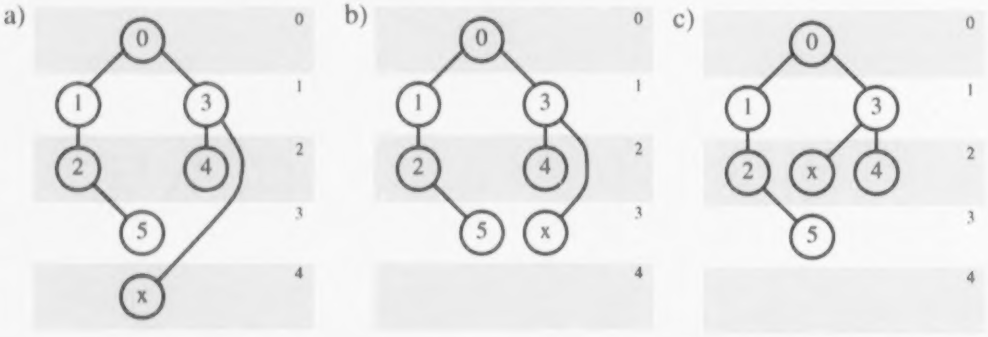


Figure 4.10 Node migration. (a) original annotated net from makenet, (b) one migration of node *x*, (c) a second migration to a stable execution step

It may happen that no nodes migrate at all during this analysis phase. This simply indicates that there is no parallelism that can be exploited based on the dependency edges that were found. `nodemig` will output a message if this occurs since this will likely affect the user's choice of coalescing parameters in the following analysis phase.

4.5.1 `nodemig` Invocation Syntax

`nodemig` is invoked from the command-line with the following syntax:

```
nodemig <source net name> <target net name>
```

where the source net is the net description file produced from `makenet` and the target net is the net description generated following the analysis. Any errors or anomalies that occur during node migration will result in the generation of an appropriate error message and the immediate termination of the analysis process.

Chapter 5

Program Partitioning Via Node Coalescing

5.1 Introduction

The previous chapter laid the foundation necessary for partitioning a sequential program into parallel subtasks. This chapter describes how these results are used to safely guide the partitioning process. The final stage in NetGen's analysis is *node coalescing*. Node coalescing begins to group nodes into clusters (called supernodes) that represent segments of code that should be executed sequentially. The criteria used to select statements to be coalesced are supplied by the researcher. Typically, this involves choosing a particular node coalescing technique and by varying several cluster-forming parameters. The final net will consist of a number of these coalesced clusters, some of which may be executed in parallel.

It is this stage of the parallelization process that the researcher has the most influence in affecting the program parallelization decisions. The effects of these decisions on the final program partitioning is a subject for further empirical study beyond this thesis.

This chapter describes `coalesce`, the tool used to partition a net into supernodes. As illustrated in Figure 5.1, `coalesce` requires the net description produced from the node migration stage, a file containing coalescing parameter values, and the user's choice of a coalescing technique. A coalesced net description is produced. The final section of this chapter discusses the intermediate net description language that the NetGen modules use to communicate results. A perspective on some implementation issues for the NetGen project is also provided.

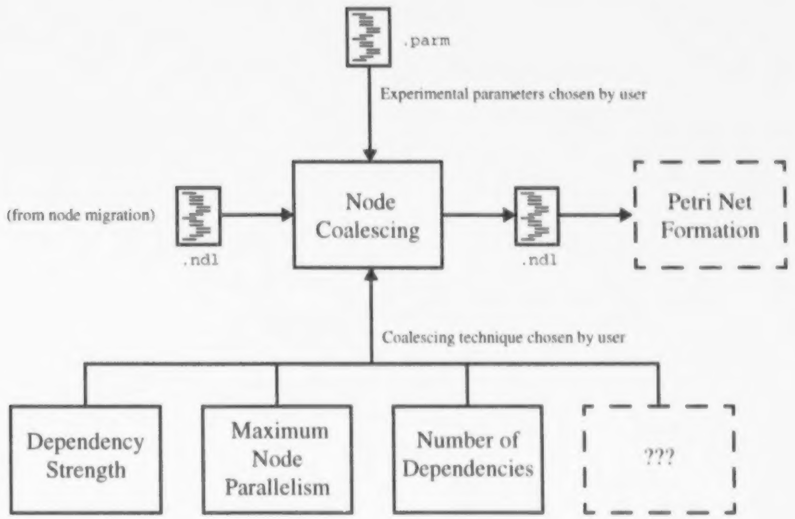

.parm
Experimental parameters chosen by user
(from node migration)
.ndl
Node Coalescing
.ndl
Petri Net Formation
Coalescing technique chosen by user
Dependency Strength
Maximum Node Parallelism
Number of Dependencies
???

Figure 5.1 Input/output requirements of the node coalescing phase

5.2 Supernodes

The fundamental building block of the final net is the *supernode*. Supernodes contain nodes that have a particular dependency relationship that the researcher wishes to explore. Well-defined criteria are used to select statements that possess these dependency characteristics. At present, these criteria include selecting nodes that have a certain "dependency strength" between them, selecting nodes based on the number of common dependencies, and choosing nodes that maximize the number of parallel nodes in the final net. The results acquired from applying these node selection techniques to a net can be modified through the variation of supernode-forming parameters by the researcher.

Supernodes are formed by coalescing statement nodes within the nets produced by the node migration phase. The constituent nodes of a supernode must be executed sequentially. In other words, a supernode contains nodes that cannot be executed in parallel based on the criteria specified by the researcher.

The number of supernodes that will be produced for a given dependency net cannot be accurately predicted beforehand. Supernode formation depends not only on the parameters specified by the user but on how well the dependencies within the net fit these criteria. Hence, grouping is influenced by both the user-specified parameters and the unique dependency characteristics of the particular net.

5.3 Supernode Properties

All supernodes possess common characteristics regardless of how their constituent nodes were chosen. As a result, once all supernodes have been formed they can be treated identically. All supernodes have the following properties:

- *A dependency net must contain at least one supernode.* Every node in the dependency net could be coalesced into a single supernode (i.e., the program will execute as it would on a sequential machine). The maximum number of supernodes is limited by the number of nodes there are in the dependency net (i.e., each node could form a separate supernode).

- *Every supernode contains at least one node.* There is no limit to the number of nodes that may belong to a single supernode.

- *A supernode may contain multiple control flow entry points and multiple control flow exit points.* Every supernode contains at least one entry node where the flow of control can enter and at least one node (possible the same node) where it can exit. All entry and exit nodes must be clearly identified. The number of entry and exit points in a supernode is dependent on the supernode formation technique.

- *Supernodes contain nodes that will be executed sequentially.* Actually, all nodes on a control path from an entry point to an exit point will be executed sequentially. There may be several such paths within a single supernode but only one will ever be executing at some instant.

- *Supernodes occupy execution steps.* A supernode can begin executing in the earliest execution step of all of its constituent nodes. It is assumed that the entire supernode will be executed in that execution step. Like their counterparts in the original dependency nets, execution steps specify the order in which supernodes must be executed.

- *Supernodes that share an execution step can be executed in parallel.* This property allows supernodes to be executed in parallel if they do not have any dependencies between them.

Each supernode is assigned a unique supernode number. This identification begins at zero and is unique only to the current net. Every supernode must retain the following information: the supernode number, the execution step, the constituent nodes, the set of entry nodes, and the set of exit nodes. Any extra information required to translate a net of supernodes into a Petri net representation can be obtained from the original net itself. For example, the ordering of the nodes within a supernode does not have to be specified since it can be determined from the node placement in the net produced from node migration.

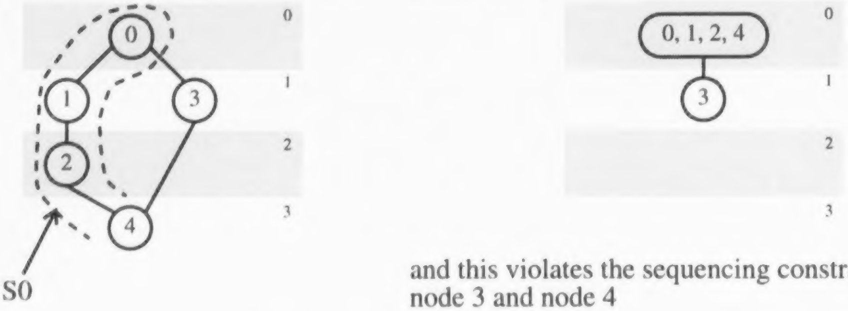
5.4 Supernode Formation Rules

Although the method of choosing nodes to be included in a particular supernode can be varied, all coalescing methods must ensure that fundamental supernode formation rules are obeyed. These rules prevent nodes from grouping if they violate necessary dependency or sequencing relationships between the statements. The basic rules that govern supernode formation are as follows:

- A node can belong to only one supernode. This rule is included simply as a formality. At present, assigning a node to multiple supernodes does not have any meaning.

- A node can be added to any supernode *except* if the following is true:

  A node n in execution step x cannot be added to a supernode S beginning in execution step e if there is some node m in execution step y such that $m \notin S$, $e \leq y < x$, and n is the next node to execute following m. This avoids two problematic situations. First, it precludes the addition of a node to a supernode if a necessary dependency would be violated by doing so. Consider the dependency net fragment in Figure 5.2. Supernode S0 wants to coalesce with node 4. However, if this were to occur then node 4 would be executed in execution step 0. Since it has a dependency with node 3 (which executes in execution step 1) then the inclusion of node 4 in S0 would clearly violate a necessary sequencing requirement (node 4 must be executed after node 3). As a result, such coalescing is not permitted.


S0 wants to coalesce with node 4:

Doing so would have node 4 execute before node 3:

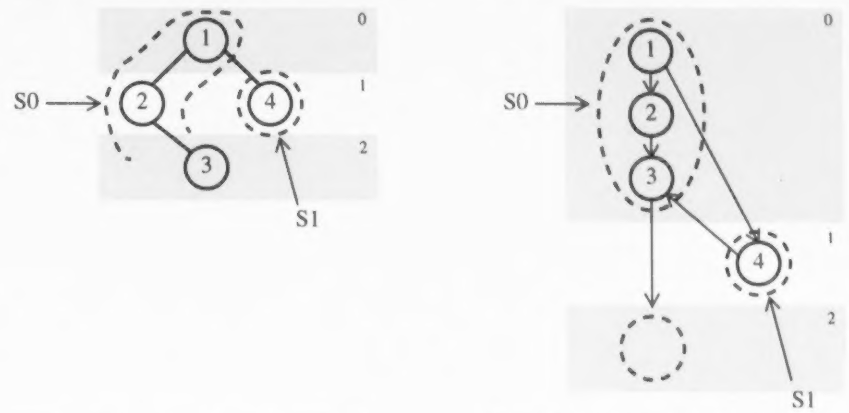
and this violates the sequencing constraint between node 3 and node 4

Figure 5.2 Node coalescing violating a necessary dependency

Second, this rule prevents the more general situation where control could leave a supernode, enter another supernode, and then return to the original supernode. This scenario is illustrated in Figure 5.3. Supernode S0 is attempting to coalesce with node 3. However, by including node 3 in S0, the flow of control would be as follows: S0 would begin executing in execution step 0, pass control to S1 in execution step 1 after node 1, and then return to S0 in execution step 0 after S1 has finished executing.

Although this flow of control can be represented in a Petri net format, the efficiency of such an approach is questioned. After control has passed from S1 back to S0, a back edge would have to be inserted into the final net to accommodate this scenario. Hence, to prevent such inefficiencies from occurring, this situation is disallowed at this stage of the analysis.

- A supernode cannot coalesce with another supernode. The inclusion of this feature at this point in the NetGen project would be premature. The ability to coalesce supernodes further implies a greater understanding of the dependency properties that caused



S0 wants to coalesce with node 3:

If S0 were allowed to coalesce with node 3:

⟶ represent flow of control

the awkward control flow shown would be required in the final Petri net representation.

Figure 5.3 Example of inefficient node coalescing.

them to be formed initially. Consequently, although the researcher may have some idea as to how and why certain supernodes should be grouped together, having NetGen perform this coalescing without some insight into the purpose is difficult to express at present. This point is discussed further in the context of future enhancements to this project in Section 7.2.2.

5.5 Node Coalescing Techniques

Now that the properties and rules for forming supernodes have been established, the procedures for selecting nodes in the net to be grouped together can be defined. NetGen provides three methods for coalescing nodes: grouping based on the strength of the dependencies between nodes, grouping based on the number of dependencies between nodes, and grouping for maximal node parallelism. Each will be discussed in detail in the following sections.

5.5.1 Node Coalescing Via A Dependency Strength Threshold

Dependencies between statements imply they cannot be executed in parallel. As a result, these statements can be grouped together and executed sequentially. However, determining how to group statements with dependencies together poses a challenging problem.

When programs are split for parallelization, the additional overhead required to preserve the context of a segment of code may be expensive. This is especially true for local variables as they are created dynamically on a stack and their allocation, use, and deallocation requires stack management. On the other hand, global variables remain in a fixed location and no management of the data area is required.

The *dependency strength* between two nodes is an empirical measure of the binding of the nodes based on the locality of the common variables between them. The strength of the dependency attempts to represent the cost of partitioning between two statements. A higher dependency strength implies that a greater overhead cost would be incurred if the two statements were placed into separate code segments. Similarly, a lower dependency strength implies a less severe cost of partitioning. Dependency strength is based entirely on data dependencies—control and I/O dependencies between statements do not affect its calculation.

With a block structured language such as Pascal, several variables in different scopes may be visible to a procedure. In general, statements that share relatively local variables (or even processor registers in some languages) have a much stronger dependency than if they shared only global variables.

To be useful, the strength of a dependency between two nodes must be quantified. This is an open problem since there are several ways of computing a value for a dependency strength. It is unclear how to determine exactly which computation methods are better or more accurate than others. However, all calculations must convey the notion of a weaker node dependency for common global variables and a stronger node dependency for common local variables.

The dependency strength concept can be used as a criterion for grouping nodes in a dependency net. This method allows nodes that have a dependency strength greater than some arbitrary threshold to coalesce into the same supernode. Nodes are left alone if they do not meet the threshold grouping requirements. The *coalescing threshold*, therefore, is defined as the minimum dependency strength between two nodes that will cause them to be coalesced into the same supernode. Note that meeting the coalescing threshold is only one requirement for aggregating the nodes. If grouping the nodes violates a fundamental supernode formation rule then they will not be coalesced, regardless of the dependence strength.

NetGen provides a node coalescing method based on the dependency strength between nodes. The researcher can experiment with the effects of varying the coalescing threshold on the supernodes formed in the final net. In addition, NetGen allows the user to associate a pseudo-dependency strength between nodes with control and I/O dependencies to provide opportunities for further experimentation.

5.5.1.1 Dependency Strength Calculation

To compute the dependency strength between two nodes requires that the common variables and their block regions be known. This information can be determined from the information stored by each node and by using variable declaration data. NetGen uses the following formula to quantify the dependency strength between two nodes:

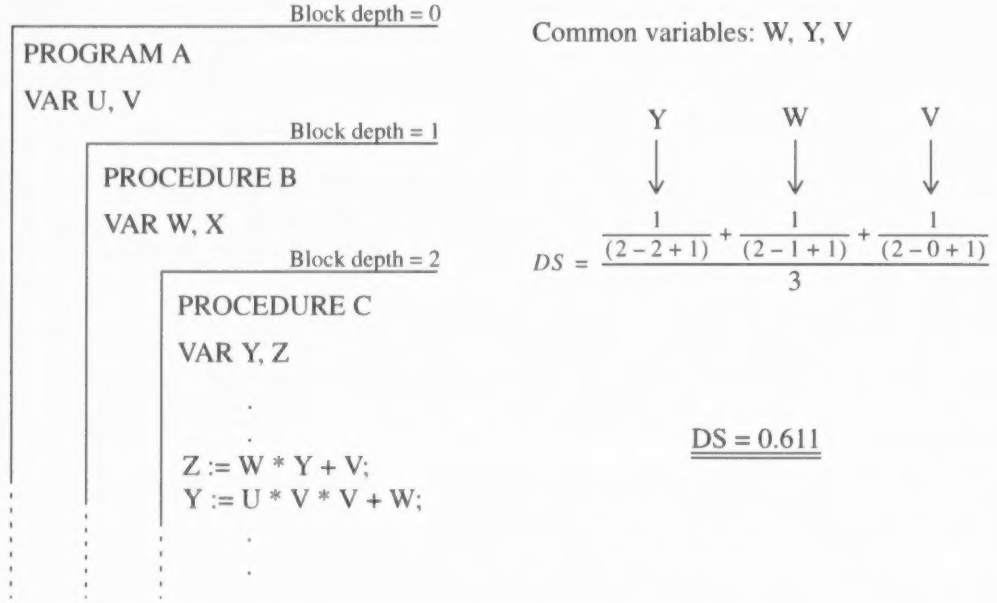
$$DS = \frac{\sum_{n} \frac{1}{BDp - BDv + 1}}{n} + \frac{\sum_{m} PD}{m}$$

where:

DS = dependency strength
BDp = block depth of the current net in which the two statements are found
BDv = block depth of the common variable
PD = a pseudo-dependency parameter (discussed in Section 5.5.1.2)
n = the number of common variables between the statements
m = the number of control and I/O dependencies between the statements (used with pseudo-dependencies)

This formula is unique to the NetGen project and produces a relatively higher value for stronger dependency strengths and a relatively lower value for weaker dependency strengths. Each common variable is represented only once in this calculation. If there are no common variables between nodes then the dependency strength is zero. As can be seen, this formula computes an average dependency strength using all common variables between two nodes. The average was included to alleviate the problem of a single variable influencing the dependency strength result. For example, if two statements have five global variables in common and only one local variable in common then the binding should be made relatively weak between them. The single local variable should not have very much effect on the dependency strength calculation.

Figure 5.4 demonstrates how the dependency strength is calculated between two nodes (without pseudo-dependencies).



```
PROGRAM A
VAR U, V
    PROCEDURE B
    VAR W, X
        PROCEDURE C
        VAR Y, Z
            .
            .
        Z := W * Y + V;
        Y := U * V * V + W;
            .
            .
```

Common variables: W, Y, V

$$DS = \frac{\overset{Y}{\overset{\downarrow}{\frac{1}{(2-2+1)}}} + \overset{W}{\overset{\downarrow}{\frac{1}{(2-1+1)}}} + \overset{V}{\overset{\downarrow}{\frac{1}{(2-0+1)}}}}{3}$$

$$\underline{\underline{DS = 0.611}}$$

Figure 5.4 Calculation of dependency strength between statements.

5.5.1.2 Pseudo-Dependency Strengths

In addition to the dependency strengths that are calculated automatically between nodes that share a common variable and a data dependency, the user can also direct NetGen to add a specified amount to the dependency strength between statements to account for control and I/O dependencies. Table 5.1 describes three parameters available to the researcher that can provide pseudo-dependency strengths. The pseudo-dependency parameters default to zero and can be changed by specifying their new value in the node coalescing parameters file. When a control or I/O dependency is detected between nodes then the value specified by the corresponding parameter will be averaged into the dependency strength calculation. If there are no such dependencies between nodes then the pseudo-dependency strength is not considered in the calculation of the dependency strength..

Parameter	**Description**
PSEUDO_CONTROL_T	Dependency strength between a node and any node control dependent on it via a True edge
PSEUDO_CONTROL_F	Dependency strength between a node and any node control dependent on it via a False edge
PSEUDO_IO	Dependency strength between two nodes that share an I/O dependency

Table 5.1 - Pseudo-dependency parameters

The PSEUDO_CONTROL_T and PSEUDO_CONTROL_F parameters are used to assign different pseudo-control strengths to the True and False paths following a condition or loop header node. This can potentially be used to force nodes to coalesce along one of the branches by providing a relatively large pseudo-control parameter for the desired branch. The PSEUDO_IO parameter is used to assign a dependency strength between nodes that have an I/O dependency.

5.5.1.3 Computing Target Nodes

In some of the algorithms presented in this section, it is important to determine what nodes can be executed immediately following a particular node (referred to as the *targets* of the node). This information is summarized in the Target(n) set, which is derived entirely from the dependency net. The algorithm used to compute the Target(n) set is described in

Appendix B.4.

The algorithm is divided into three separate cases: determining the next node following a condition statement, determining the next node following a loop header, and determining the next node following any other statement type. Each case is described below.

There are two types of condition statements: those with a single True branch and those with both True and False branches. Both types require special attention when computing the set of nodes that can execute after a predicate node. In the latter case, the next nodes to execute are those at the start of the True and False branches. The determination of these nodes requires an examination of the execution step following the predicate node for the nodes that are control dependent on the predicate node (lines 2-8). In the former case, the nodes in the next execution step that are control dependent on the predicate node and the first nodes reachable after the condition region must be added to the Target(n) set (lines 9-16). The algorithm uses a variable *found_false* to indicate whether or not a False branch was found after the predicate node.

Loop headers have two possible exit points: the beginning of the loop body or the first statement following the loop. Like the True branch in the predicate node case discussed above, all nodes in the execution step following the loop header that are control dependent on the loop header are added to the Target set (lines 18-20). In addition, the first nodes that are reachable in an outer dependency region (i.e., outside the loop body) are also added to the set (lines 21-27).

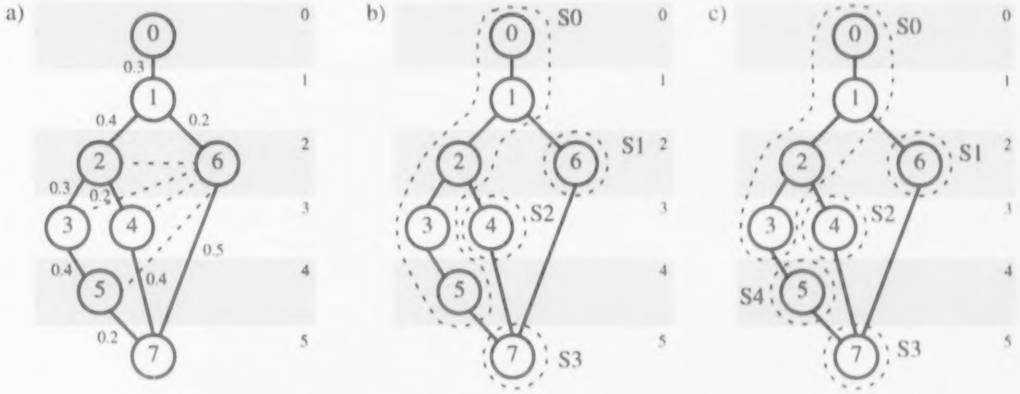
For any other type of node, the set of nodes to be executed next consist of the first nodes in a subsequent execution step that do not have a soft dependency with the subject node (line 30). If the next node lies in an outer dependency region (indicating that the subject node is the last node in a loop iteration) then the region header node is added to the Target set (line 34).

5.5.1.4 Coalescing Threshold

The coalescing threshold for node grouping is specified in the coalescing parameters file with the COALESCING_THRESHOLD parameter. The default coalescing threshold is zero.

5.5.1.5 Node Coalescing Algorithm

The algorithm used to coalesce nodes based on the dependency strength characteristics between them consists of several parts. Some parts are responsible for creating supernodes while others ensure that the fundamental supernode formation rules are obeyed and that all necessary supernode information is retained. The algorithm is presented in Appendix B.5 in four parts. For each net to coalesce, the algorithm's parts should be applied in the order they appear in the Appendix. The various parts of the algorithm will be discussed next and applied to the example dependency net in Figure 5.5a.



d) For S0, Entry = 0, Exit = 1, 2, 3
For S1, Entry = 6, Exit = 6
For S2, Entry = 4, Exit = 4
For S3, Entry = 7, Exit = 7
For S4, Entry = 5, Exit = 5

Figure 5.5 Node coalescing using dependency strength criterion. (a) sample dependency net, (b) after supernode formation application, (c) after supernode splitting, and (d) the entry/exit nodes

The first part of the algorithm simply traverses the nodes of the net and forms supernodes based on the dependency threshold criterion. Only minor checking is done in this part to ensure that the node formation rules are obeyed. Each execution step in the net is visited in order (line 2). A new supernode is created for each node in the current execution step that does not already belong to a supernode (lines 3-5). The algorithm will then locate all nodes in the net that can coalesce into that supernode. After the first node has been placed in the supernode, a search is initiated to find all nodes within the current and next execution steps (line 9). All nodes within that range are also candidates for coalescing. Nodes within the current execution step are checked because even though they could potentially be executed in parallel, based on the coalescing parameters specified by the researcher it may be better to group them into the same supernode. Of all possible nodes that can be executed next, only those that lie in the same dependency region, are not already members of a supernode, have a dependency strength greater than the coalescing threshold, do not have any soft dependencies with the subject node, and that lie within the current condition region will be added to the supernode (line 10). The soft dependency criterion was inserted because grouping nodes with soft dependencies is not really an important efficiency issue. Since the nodes separated by a soft dependency will never be executed in the same control flow (i.e., never executed truly in parallel) then grouping them into the same supernode to reduce potential context switching costs between them is irrelevant. The condition regions criterion was added to ensure that a node outside a condition statement would not be coalesced with only a single branch of the conditional (thereby averting the situation where two distinct branches would compete over coalescing the same node). Newly added nodes are subject to this same procedure applied to them. Once there are no more nodes found that meet the coalescing criterion in line 10, the next supernode is started and this process is repeated.

Figure 5.5b illustrates the application of this first part of the coalescing algorithm to the example net. The coalescing radius is 0.3. Note that node 6 has not been coalesced with any other node, even though it has a dependency to node 7 that is above the coalescing threshold. Node 7 is rejected because it does not lie in the subsequent execution step after node 6 and it is in a different condition region than node 6 (i.e., there is no soft dependency between node 6 and node 7).

The second part of the algorithm ensures that the supernodes formed in the previous step obey all supernode formation rules. Now that every node in the net belongs to a supernode, the nodes are checked again to ascertain whether or not a particular node could have been coalesced into more than one supernode (line 5). If a node has a choice of coalescing with multiple supernodes then the node and its targets will be split from the supernode in which it is presently contained and will form a new supernode. This is done primarily in fairness for the other supernodes that want that node.

In addition, if the target of a node in one execution step is to a node in a previous execution step then the target node is split from its supernode and placed in a new supernode (line 7). This is done to eliminate the possibility of control flow leaving a supernode, entering another, and then returning to the first. Figure 5.5c illustrates the application of this second part of the coalescing algorithm to the example net. Note that nodes 4 and 5 now lie in distinct supernodes, by the condition on line 7.

The next stage of the algorithm "compresses" the net containing the supernodes by removing any empty execution steps. The fourth part of the algorithm determines the entry and exit nodes of each supernode. It is those nodes where the flow of control will enter and leave, respectively. The NET_INITIALIZE node is always considered an Entry node and, similarly, the NET_TERMINATE node is always an Exit node (lines 1-2). Once again the target of each node in all of the supernodes is determined (lines 3-5). If the target node lies outside of the supernode then the source node is marked as an Exit and the target node is marked as an Entry. The results of this algorithm applied to the example are shown in Figure 5.5d.

5.5.2 Coalescing Via Number Of Dependencies Threshold

The algorithm presented in the previous section can be adapted somewhat to provide an alternative criterion for coalescing nodes. Instead of coalescing based on the strength of dependencies between nodes, nodes are now aggregated by the number of common dependencies they share. The total number of dependencies is computed by summing the number of variables the two nodes have in common and any control or I/O dependencies. The *number of dependencies threshold* for this technique refers to the minimum number of dependencies that must exist between two nodes that will cause them to be coalesced into the same supernode. The value of this threshold can be specified by the NDEP_THRESHOLD parameter in the node coalescing parameters file.

The algorithm for this coalescing method is presented in Appendix B.6. It is identical to the algorithm presented in the previous section with the exception that instead of checking dependency strengths between nodes to determine coalescing, the number of dependencies is checked. This affects line 10 in the first part of the algorithm and line 5 in the second.

5.5.3 Coalescing For Maximal Node Parallelism

An alternative method of grouping nodes is based on exploiting the maximum possible parallelism that has been detected for a sequential program. This essentially forms any nodes that exhibit parallelism in the dependency net into a separate supernode. Segments of sequential execution are also coalesced into a single supernode. This technique differs from the previous two in that all the supernodes formed will be single entry, single exit. This alleviates many of the coalescing problems of the previous approaches.

The algorithm for coalescing for maximal node parallelism is specified in Appendix B.7. It consists of two stages: a supernode formation stage and a net compression stage. Supernode formation involves identifying segments of nodes in the net such that the segment has a single entry point and a single exit point. In other words, the nodes within the supernode can be executed sequentially. These are identified (in part) by checking the fan-out of a node (lines 13-17). If the flow of control can branch to more than one node then each of those target nodes will begin a new supernode and the current node will terminate whatever supernode it is a member. Similarly, if a node has multiple control flow paths entering it then it is made the start of a new supernode (lines 18-27). Since supernodes have only one flow of control through them, the Entry and Exit nodes are easily identified.

This coalescing technique does not have any user-defined parameters as its coalescing criterion is clearly specified. An example dependency net that is coalesced with this method is illustrated in Figure 5.6.

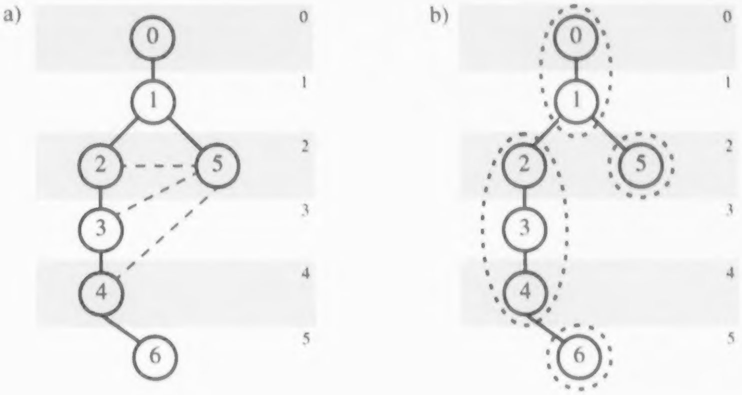


Figure 5.6 Node coalescing for maximal node parallelism. (a) Sample dependency net, and (b) resulting node groupings

5.6 `coalesce` Invocation Syntax

The partitioning module is invoked from the command-line with the following syntax:

```
coalesce [-cr] [-nd] [-md] -p <parameter file> <input net> <output net>
```

where:

-cr selects node grouping based on the coalescing radius approach
-nd selects node grouping based on the number of dependencies
-md selects node grouping based on sequential dependencies

The node coalescing parameter file contains a set of parameter-value pairs. Each parameter is specified on a separate line in the file using the syntax:

```
<PARAMETER NAME> <VALUE>
```

A sample parameter file is provided in Appendix C.2.

The parameter file, one coalescing technique, the source net, and the file name of the new coalesced net must be specified. Any errors or anomalies that may occur during this phase will result in the generation of an error message and the immediate termination of the module.

5.7 Net Description Language

Following each NetGen module, the results of the analysis performed must be communicated to the other modules. Since these other modules exist as separate executable files, a net format that is both persistent (i.e., exists even after a module has completed) and descriptive is required. The *net description language* is a method of describing the structure of a dependency net in a textual format. Although it is intended to be analyzed and generated by other modules, the format of the net description file is presented in a human-readable form.

A net description file is composed of two sections: variable declarations and net descriptions. The variable declarations section summarizes all variables that have been declared in the program. Variable names are stored along with their declaration numbers and declaration block depths. Aliasing information is not required since this information has already been added to the *def* and *use* sets of each statement (which are described in this file). A detailed description of the dependency nets are included following the declaration information. The program block net is stored first followed by any procedure or function block nets. The nets are stored in the order in which their blocks appeared in the original Pascal source. Each net is described by listing its constituent execution steps in order and providing detailed information about each node contained therein. All data items specified in Table 4.2 for a node are represented in the net description file in a textual format. Supernode descriptions are also included in the appropriate execution step. Supernodes are described by indicating the supernode number, all constituent nodes, and all entry and exit nodes.

5.7.1 Language Description

The complete context-free grammar of the net description language can be found in

Appendix A.2. A sample net description file that illustrates the contents of the various sections is provided in Appendix C.

5.8 NetGen Implementation

5.8.1 Development Tools

NetGen was developed on a SPARC workstation under the Unix operating system and was implemented in strict ANSI C using the GNU gcc compiler version 2.3.3. Early versions of the project were developed on a Pentium-based workstation running the Windows NT operating system.

The lexical analyzer used for the parsing of the sequential program was generated by GNU Flex version 2.5.1. The Pascal language parser was generated using GNU Bison version 1.22. Flex takes a description of the tokens to look for and produces a "lexer" that can identify those tokens in an input stream. Bison takes an annotated grammar description of a language and produces a parser that can recognize certain language elements (e.g., statements).

5.8.2 Correctness Of Design

No formal proofs of correctness were performed on the code developed for this project. Similarly, formal methods were not used to verify the correctness of the many algorithms developed in this thesis. Instead, a more intuitive and deductive approach was taken when justifying the procedures and results produced by the various algorithm designs. The algorithms developed in this thesis are not overly complex and such a verification method was deemed sufficient. Nonetheless, extensive testing of the various modules with several net structures was performed successfully and the results verified manually.

5.8.3 Design Safety

An important consideration when developing a tool to parallelize sequential programs is that of partitioning correctness. The partitioned software should produce the same results as the original sequential program. To this end, safety features are built into the analysis process to ensure a correct program partitioning. If there is ever any doubt as to whether or not a dependency exists between two statements, the conservative approach is taken and one is assumed. This may be an incorrect assumption but it is a safe assumption that guarantees a correct partitioning. A similar scenario exists when computing aliases, where aliases are assumed in the presence of any doubt.

Chapter 6

Program Partitioning With NetGen

6.1 Introduction

The preceding chapters described a tool to experiment with the partitioning of a sequential program subject to several parallelization parameters. This chapter will provide a demonstration of partitioning a sequential test program using the techniques developed in the preceding chapter. This chapter merely demonstrates the analysis capabilities of NetGen. Interpretation of the results is not one of the mandates of this thesis.

6.2 Sample Program `analysis`

6.2.1 Program Description

The sample program presented in Appendix C.3 will be used to demonstrate the analysis techniques in this section. This program was adapted from a similar program presented in [Etter88] on page ANS-34. The program reads 61 population values and determines the years of greatest percentage increase in population. It also computes the average increase in population per year for the total period of time.

6.2.2 Node Coalescing

Node coalescing via the dependency strength criterion with a coalescing threshold of 1.0 yields the partitioning shown in Table 6.1. Node coalescing via the number of dependencies criterion with a number of dependencies threshold of 1.0 yields the partitioning shown in Table 6.2. Finally, node coalescing via the maximal node parallelism criterion yields the partitioning shown in Table 6.3.

6.3 Sample Program `leastsquares`

6.3.1 Program Description

This sample program demonstrates the calculation of the coefficients of a straight line that

Supernode Number	Execution Step	Constituent Nodes	Entry Nodes	Exit Nodes
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	2	3	3	3
4	3	4,5,6,7,9,10	4	6, 7
5	4	8	8	8
6	5	11, 12, 13	11, 12	13
7	6	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.1 - Program analysis Coalescing Via Dependency Strength (CR = 1.0)

Supernode Number	Execution Step	Constituent Nodes	Entry Nodes	Exit Nodes
0	0	NET_INITIALIZE, 1, 2	NET_INITIALIZE	1, 2
1	1	3	3	3
2	2	4,5,6,7,9,10	4	7
3	3	8	8	8
4	4	11, 12, 13	11, 12	13
5	5	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.2 - Program analysis Coalescing Via Number Of Dependencies (ND = 1.0)

is fit to a regression of points (least squares fit). The points in the curve are supplied by the user. The source listing for this example is provided in Appendix C.4 and a sample net description file can be found in Appendix C.1. The program was adapted from program 12.7 in [McWa83].

6.3.2 Node Coalescing

Table 6.4 illustrates the coalescing of program `leastsquares` via a dependency strength criterion with a coalescing threshold of 1.0. However, regardless of the coalescing threshold (unless CT = 0) all node groupings will be the same if all variables are declared in the same block. This is to be expected since a dependency strength varies across statements that share variables declared in different blocks. This clustering can be

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	2	3, 4	3	4
4	3	5	5	5
5	3	9	9	9
6	4	6	6	6
7	4	10	10	10
8	5	7	7	7
9	5	8	8	8
10	6	11	11	11
11	6	12	12	12
12	7	13	13	13
13	8	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.3 - Program analysis Coalescing For Maximal Node Parallelism

influenced by modifying the pseudo-dependencies. Table 6.5 illustrates the effect of setting the PSEUDO_CONTROL_T parameter to 1.5.

Table 6.6 and Table 6.7 show the resulting supernodes when a coalescing based on the number of dependencies is performed. The number of dependencies thresholds are 1.0 and 2.0, respectively. Finally, Table 6.8 illustrates the coalescing of nodes in the sample program for maximal node parallelism.

6.4 Sample Program `Graphics`

6.4.1 Program Description

This sample program demonstrates vector plotting on a character device by drawing a square spiral rotated through 45 degrees. The source listing can be found in Appendix C.5. Note that the numbers located in the left margin of the program listing refer to the node numbers that are assigned during program analysis. Although there are several pro-

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	2	5	5	5
6	3	6, 7, 8, 9, 10	6	7, 8, 9, 10
7	4	11, 12, 13	11	13
8	5	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.4 - Program leastsquares Coalescing Via Dependency Strength (CT = 1.0)

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE, 1, 2, 3, 4	NET_INITIALIZE	1, 2, 3, 4
1	1	5, 6	5	6
2	2	7	7	7
3	2	8	8	8
4	2	9	9	9
5	2	10	10	10
6	3	11	11	11
7	4	12	12	12
8	5	13	13	13
9	6	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.5 - Program leastsquares Coalescing Via Dependency Strength (CT = 1.0, PSEUDO_CONTROL_T = 1.5)

cedures involved, only the non-trivial main program block will be examined in this section. This program was adapted from Program 10.4 in [McWa83].

6.4.2 Node Coalescing

Table 6.9 illustrates the coalescing of program `graphics` via a dependency strength cri-

Supernode Number	Execution Step	Constituent Nodes	Entry Nodes	Exit Nodes
0	0	NET_INITIALIZE, 1, 2, 3, 4	NET_INITIALIZE	1, 2, 3, 4
1	1	5, 6, 7, 8, 9, 10	5	5
2	2	11, 12, 13	11	13
3	3	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.6 - Program leastsquares Coalescing Via Number Of Dependencies (ND = 1.0)

Supernode Number	Execution Step	Constituent Nodes	Entry Nodes	Exit Nodes
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	2	5	5	5
6	3	6, 9	6	6, 9
7	4	7	7	7
8	4	8	8	8
9	4	10	10	10
10	5	11, 12	11	12
11	6	13	13	13
12	7	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.7 - Program leastsquares Coalescing Via Number Of Dependencies (ND = 2.0)

terion with a coalescing threshold of 1.0. Table 6.10 shows the resulting supernodes when a coalescing based on the number of dependencies is performed. The number of dependencies threshold is 2.0. Finally, Table 6.11 illustrates the coalescing of nodes in the sample program for maximal node parallelism.

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	2	5	5	5
6	3	6	6	6
7	4	7	7	7
8	4	8	8	8
9	4	9	9	9
10	4	10	10	10
11	5	11, 12, 13	11	13
12	6	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.8 - Program leastsquares Coalescing For Maximal Node Parallelism

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	1	5	5	5
6	1	6	6	6
7	1	7	7	7
8	2	8	8	8
9	3	9	9	9
10	4	10	10	10
11	4	11,12	11,12	11,12
12	4	13	13	13
13	5	14	14	14
14	5	15,16	15,16	15,16
15	5	17	17	17
16	6	18	18	18
17	6	19,20	19,20	19,20
18	6	21	21	21
19	7	22	22	22
20	7	23,24	23,24	23,24
21	8	25	25	25
22	8	26	26	26
23	9	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.9 - Program graphics Coalescing Via Dependency Strength (CT = 1.0)

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	1	5	5	5
6	1	6	6	6
7	1	7	7	7
8	2	8	8	8
9	3	9, 10, 13, 14, 17, 18, 21, 22	9	10, 14, 18, 22
10	4	11	11	11
11	4	12	12	12
12	5	15	15	15
13	5	16	16	16
14	6	19	19	19
15	6	20	20	20
16	7	23	23	23
17	7	24	24	24
18	8	25	25	25
19	8	26	26	26
20	9	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.10 - Program graphics Coalescing Via Number Of Dependencies (ND = 2.0)

Supernode Number	**Execution Step**	**Constituent Nodes**	**Entry Nodes**	**Exit Nodes**
0	0	NET_INITIALIZE	NET_INITIALIZE	NET_INITIALIZE
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	1	5	5	5
6	1	6	6	6
7	1	7	7	7
8	2	8	8	8
9	3	9	9	9
10	4	10	10	10
11	4	11	11	11
12	4	12	12	12
13	4	13	13	13
14	5	14	14	14
15	5	15	15	15
16	5	16	16	16
17	5	17	17	17
18	6	18	18	18
19	6	19	19	19
20	6	20	20	20
21	6	21	21	21
22	7	22	22	22
23	7	23	23	23
24	7	24	24	24
25	8	25	25	25
26	8	26	26	26
27	9	NET_TERMINATE	NET_TERMINATE	NET_TERMINATE

Table 6.11 - Program graphics Coalescing For Maximal Node Parallelism

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis described the development of an experimental tool that partitions software for net-based computing environments. Net-based computing models the execution of software as a Petri net, where segments of sequential code represent places and transitions coordinate the execution of such segments. This model allows the representation of synchronization, sequentiality, and parallelism between segments of code.

The tool (NetGen) bases its parallelization decisions on the results of a static dependency analysis of a sequential program. A dependency net structure is produced that summarizes the dependencies between statements and their required execution order. The partitioning process groups (or coalesces) nodes in the net based on certain dependency criteria defined by the researcher. There are several parameters and coalescing techniques available to experiment with. The effects of these parameters on the resulting net structures is a subject for further investigation.

The thesis was divided into three main sections. First, background material pertinent to the program parallelization process was presented. In the course of this discussion, several relevant examples of other program parallelization tools were described. Second, the NetGen tools that transform a sequential program into an intermediate net format prior to partitioning were discussed. Finally, the node coalescing techniques and the results of several sequential program partitionings were presented.

The NetGen project provides a method of experimentally generating net-based programs from existing programs. An evaluation of the partitioning results is beyond the scope of this thesis. This leaves several related open problems that can be studied with NetGen:

1. Identify general coalescing parameters that produce a "good" program partitioning for all classes of programs. A "good" program partitioning is one that produces a parallel program that exhibits some execution speedup over its sequential counterpart. The effectiveness of a particular program partitioning requires information about the target architecture and other partitioning results to which it can be compared.

2. Investigate what coalescing method should be applied when confronted with a particular class of programs (e.g., how should scientific programs with many loops be best coalesced).

3. Determine general architecture requirements and features based on the coalescing results of general-purpose programs (e.g., study how general-purpose programs coalesce into nets and use this information to influence the development of net-based architectures).

Any general coalescing parameters could be useful when developing compilers for the target machine. If it can be determined that a particular class of program is being compiled (such as a program with many sequential statements or condition constructs) then the node coalescing technique and corresponding parameters best suited to that type of program can be applied. In order to obtain as much information as possible for the development of future compilers, many classes of programs must be studied with several parameter values. The coalescing techniques discussed in this work attempt to solve the software partitioning problem based on the usage of local and global variables between statement, by the degree of coupling between statements, and by allowing maximal parallelism. Additional parameters could be added to the existing coalescing procedures and new supernode formation algorithms could be implemented and studied.

From a preliminary analysis of the results it would seem that this tool (in its present form) would be mainly suitable for the exploitation of finer granularities of parallelism. Such parallelism could best be exploited on superscalar processors which permit the concurrent execution of several independent instructions. This is most likely due to the analysis procedure performed by NetGen since all parallelization decisions are made based on the dependencies determined from a scalar analysis of the source. As a result, the tool could be very useful in determining parallelism in programs that will be targeted for superscalar architectures.

7.2 Future Work

As this is the first experimental parallelization tool for the net-based computing ideas under investigation at the University of Alberta, there are several directions in which Net-Gen can be further developed. This section presents many possible ideas for extending this project.

7.2.1 Petri Net Generation From A Coalesced Net

Clearly, the largest area yet to be developed is the conversion of the final coalesced net into a Petri net representation. This is an essential step in the program conversion process as the target machine will only accept programs described in a Petri net format. This format will also facilitate the evaluation of the partitioning results either by direct net execution on the target machine or via simulation. Unfortunately, the development of net-based architectures is still in the research phase. Until suitable architectures have been implemented to accommodate net-based computing, a simulator has been developed that is useful in evaluating the performance of programs represented as Petri nets [JoCa95]. The simulator allows the researcher to dynamically visualize the flow of tokens (control flow) between places (sequential code segments). Control flows representing parallelism, synchronization, and non-determinism can be observed. Several simulation parameters can be modified including the number of available processors in the target environment and the execution times of specific code segments.

Although ideally suited to make performance estimations of a net-based program on a particular architecture, this simulator was not used to study the effectiveness of the partitioning strategies described in this thesis. The reason for this is three-fold. First, the characteristics of the target machine are not well-defined and therefore the use of such a simulator is precluded. Second, in order to make performance measurements, some common point of reference is required to which the performance of a particular net structure can be compared. Since no studies in this area have been completed for net-based computing, any performance results obtained would be meaningless. Third, the mandate of this thesis was to provide a tool that allows experimental generation of net-based programs. The evaluation of the effectiveness of the individual partitioning strategies is certainly a subject for future study, but is beyond the scope of this work.

The information gathered about supernodes in the node coalescing phase will be used to generate a Petri net description of the original sequential program. Supernodes will translate into Petri net places and the location and ordering of supernodes in the final dependency net will be used to determine the necessary transitions between places.

For completeness, the sequential program described in Figure 7.1a is partitioned via NetGen and manually converted into a Petri net representation (d). The intermediate net generation stages are illustrated in parts (b) and (c).

7.2.2 Coalescing Supernodes

The supernodes formed using the techniques described in this thesis vary in size and number. Unfortunately, the size of the resulting supernodes may be inappropriate for the target machine. For example, a large number of supernodes (indicating a finer granularity) may result in context switching overheads on the target machine that preclude such a program partitioning. Although the researcher should make every effort to incorporate architecture characteristics into the choice of parameters, this may be insufficient. As a result, a further coalescing of supernodes into larger entities may be required at a later stage. Determining ways of coalescing supernodes might also provide additional insight into the features desired in a node coalescing algorithm

7.2.3 Dependency Analysis Improvements

As the effectiveness of the program parallelization is strongly dependent on the rigorousness of the initial program dependency analysis, any improvement to the dependency analysis phase is encouraged. At this point, NetGen is limited to a scalar analysis of the source program to determine dependencies. Consequently, the parallelization of any program

a)

```
        PROGRAM TRIANGLE;

        VAR a,b,c,d,e,area,perim : REAL;

        BEGIN
1          WRITELN('Enter 3 sides of a triangle');
2          READLN(a,b,c);
3          d := a*a + b*b;
4          e := c*c;
5          perim := a + b + c;
6          IF (d = e) THEN
              BEGIN
7                WRITELN('This is a right triangle');
8                area = 0.5*a*b;
9                WRITELN('area = ',area)
              END
           ELSE
10            WRITELN('Not a right triangle');

11         WRITELN('Perimeter = ',perim)
         END.
```

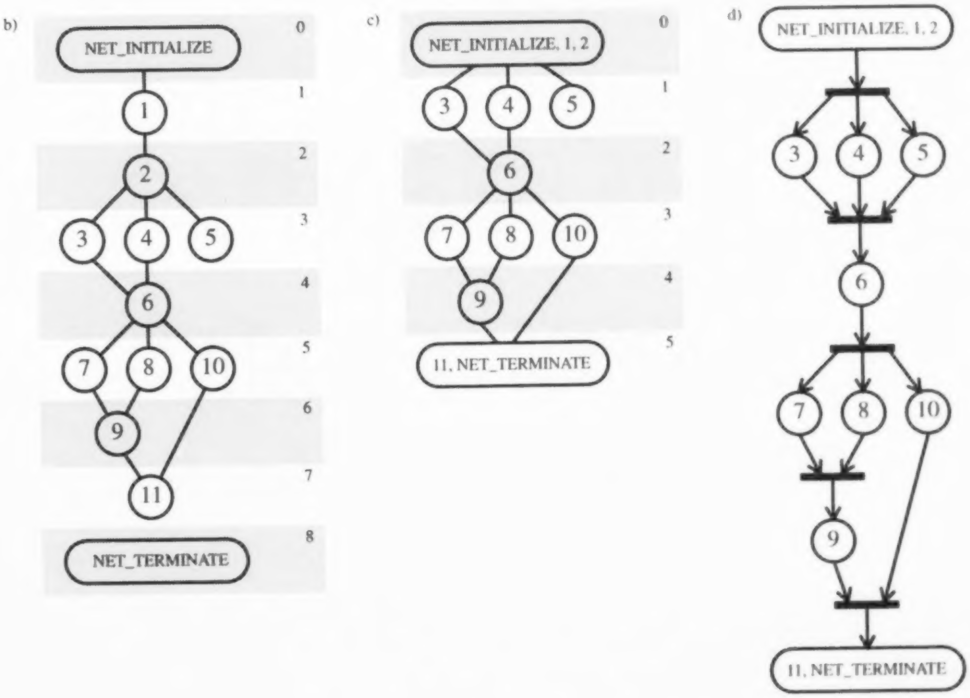

b)
NET_INITIALIZE
0
1
1
2
2
3
3 4 5
4
6
5
7 8 10
6
9
7
11
8
NET_TERMINATE
c)
NET_INITIALIZE, 1, 2
0
1
3 4 5
2
6
3
7 8 10
4
9
5
11, NET_TERMINATE
d)
NET_INITIALIZE, 1, 2
3 4 5
6
7 8 10
9
11, NET_TERMINATE

Figure 7.1 Conversion of an annotated net into a Petri net for net-based execution. (a) the program to be transformed, (b) the annotated net after node migration, (c) a possible coalesced net, and (d) the final Petri net representation.

that makes extensive use of array structures could be retarded. A scalar analysis also precludes any dependency analysis across loop iterations, which are a rich source of parallelism in many application domains. As a result, analysis of array subscripts could greatly improve the accuracy and scope of the statement dependency calculations.

An additional method that can be employed to improve the dependency analysis determination is the application of several parallelism improvement transformations on the source code. These are typically employed in other parallelization tools (e.g., PTRAN). Several approaches can be taken to eliminate dependencies and thus improve the opportunities for parallelism. Control dependencies can be eliminated using *if-conversion* [AKPW83]. This approach converts control dependencies into data dependencies by computing the exact condition required for the execution of a statement (called the mask of the statement). "The program can then be rewritten as a sequence of...statements whose execution depends on the mask; all other if-statements...are eliminated." [ZiCh90] To remove data dependencies, [Sarkar91a] suggests employing subscript value analysis, storage transformations (such as changing scalar variables into one-element arrays), and rewriting the program in a single-assignment applicative language.

7.2.4 Source Language Improvements

As described earlier, only a subset of standard Pascal was sufficient to generate programs suitable for partitioning by NetGen. Unfortunately, many useful features were eliminated from the language. To make partitioning suitable for a wider range of software applications these language elements should be included. Most importantly, pointer variables and complex data structures should be supported as they are commonplace in modern software designs. Unfortunately, the addition of such elements often proliferates the complexity of the dependence analysis.

Another useful feature would be the inclusion of additional sequential language front-ends for the analysis. Popular imperative languages such as Fortran, C, or even assembler can be provided. Unfortunately, each language provides unique parallelization challenges that must be resolved, often using techniques applicable only to that language. For example, assembler code is typically pervaded with many indirect memory references. Much like the pointer variable problem in other languages, these references must be resolved statically in order to perform an accurate dependence analysis. However, since they are used much more frequently in assembler, a large amount of the dependency analysis effort must be expended on this problem.

Bibliography

ABKP86 R. Allen, D. Baumgartner, K. Kennedy, A. Porterfield, "PTOOL: A Semi-automatic Parallel Programming Assistant", Proc. 1986 Int. Conf. on Parallel Processing, 1986, pp. 164-170.

AhSeUl86 A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

AKPW83 J. Allen, K. Kennedy, C. Porterfield, J. Warren, "Conversion of Control Dependence to Data Dependence", Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages, 1983, pp. 177-189.

AlGo89 G. S. Almasi, A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, 1989.

AmBuZi92 A. L. Ambler, M. M. Burnett, B. A. Zimmerman, "Operational Versus Definitional: A Perspective on Programming Paradigms", IEEE Computer, Vol. 25, No. 9 (September 1992), pp. 28-43.

Baldwin87 D. Baldwin. *Why We Can't Program Multiprocessors the Way We're Trying to Do It Now.* Technical Report 224, University of Rochester Computing Science, November 1987.

BoJa66 C. Böhm, G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules", Communications of the ACM, Vol. 9, No. 5 (May 1966), pp. 366-371.

Budd91 T. Budd. *An Introduction To Object-Oriented Programming*. Addison-Wesley, 1991.

Burke90 M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis", ACM Trans. on Programming Languages and Systems, Vol. 12, No. 3 (July 1990), pp. 341-395.

Carling88 A. Carling. *Parallel Processing: The Transputer and Occam*. Sigma Press, 1988.

CHHK*93 K. Cooper, M. Hall, R. Hood, K. Kennedy, et al., "The ParaScope Parallel Programming Environment", Proceedings of the IEEE, Vol. 81, No.2 (Feb. 1993), pp. 244-262.

ChWeZa90 D. Chase, M. Wegman, F. K. Zadeck, "Analysis of Pointers and Structures", Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation, Vol. 25, No. 6 (June 1990), pp. 296-310.

Cooper83 D. Cooper. *Standard Pascal User Reference Manual.* W. W. Norton & Company, Inc., 1983.

Duncan90 R. Duncan, "A Survey of Parallel Computer Architectures", IEEE Computer, Vol. 23, No. 2 (February 1990), pp. 5-16.

Ellis87 J. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1987.

Etter88 D. M. Etter, *Problem Solving In Pascal For Engineers and Scientists.* Benjamin/Cummings, 1988.

FeOtWa87 J. Ferrante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Use in Optimization", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3 (July 1987), pp. 319-349.

Flynn72 M. J. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Computers, Vol. C-21, No. 9 (Sept. 1972), pp. 948-960.

Furht94 B. Furht, "Parallel Computing: Glory and Collapse", IEEE Computer, Vol. 27, No. 11 (November 1994), pp. 74-75.

Harrison90 W. Harrison, "Tools for Multiple-CPU Environments", IEEE Software, Vol. 7 (May 1990), pp. 45-51.

HwBr84 K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw-Hill Publishing, 1984.

JoCa95 W. Joerg, K. Campbell, "PSIM - A Simulator for Concurrent Execution of Net-based Programs", IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing, May 1995, pp. 517-520.

Johnson91 M. Johnson, *Superscalar Microprocessor Design*. Prentice-Hall, 1991.

KKPLW81 D. Kuck, R. Kuhn, B. Leasure, D. Padua, M. Wolfe, “Dependence Graphs and Compiler Optimizations”, Conference Record of 8th ACM Symposium on Principles of Programming Languages, 1981, pp. 207-218.

LeEl92 T. G. Lewis, H. El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.

Lewis94 T. G. Lewis, “Where is Computing Headed?”, IEEE Computer, Vol. 27, No. 8 (August 1994), pp. 59-63.

MaJo95 D. Maier, W. Joerg, “Parallelization of Sequential Programs for Net-based Execution”, IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing, May 1995, pp. 323-326.

McWa83 J. McGregor, A. Watt. *Pascal For Science and Engineering*. Pitman Publishing, 1983.

MuJo81 S. S. Muchnick, N. D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

Murata89 T. Murata, “Petri Nets: Properties, Analysis, and Applications”, Proceedings of the IEEE, Vol. 77, No. 4 (Apr. 1989), pp. 541-580.

Ottenstein85 K. J. Ottenstein, “A Brief Survey of Implicit Parallelism Detection”, Parallel MIMD Computation: The HEP Supercomputer and Its Applications, J. S. Kowalik Ed., The MIT Press, 1985.

PGHLLS90 C. Polychronopoulos, M. Girkar, M. Haghighat, et al, “The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran”, Languages and Compilers for Parallel Computing, D. Gelernter et al Eds., The MIT Press, 1990, pp. 423-453.

Poly88 C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

Sarkar91a V. Sarkar, “PTRAN - The IBM Parallel Translation System”, Parallel Functional Languages and Compilers, Addison-Wesley, 1991, pp. 309-391.

Sarkar91b V. Sarkar, “Automatic partitioning of a program dependence graph into parallel tasks”, IBM Journal of Research and Development, Vol. 35, No. 5/6 (Sept./Nov. 1991), pp. 779-804.

Stone90 H. S. Stone. *High-Performance Computer Architecture*, 2nd Ed. Addison-Wesley, 1990.

Ullman75 J. D. Ullman, “NP-Complete Scheduling Problems”, Journal of Computer and System Sciences, 10, 1975, pp. 385-393.

Wong92 P. Wong. *The Enterprise Executive*. Technical Report TR92-13, Dept. of Computing Science, University of Alberta, Edmonton, Canada, 1992.

ZiCh90 H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, 1990.

Appendix A

Grammar Descriptions

The grammars presented in this section are all described in Backus-Naur form (BNF), a format for specifying context-free grammars. The grammars are presented with the non-terminals alphabetized. Non-terminals are represented in lowercase and terminal symbols are indicated in uppercase. The start symbol is indicated in boldface type.

The symbol '|' that appears in many of the productions represents an alternative. For example, the production:

```
A     : B
      | C
```

implies that non-terminal A can derive either B or C.

A.1 Pascal Subset Grammar

```
actual_parameter          : expression

actual_parameter_list     : ‘(‘ actual_parm_list ‘)’

actual_parm_list          : actual_parm_list ‘,’ actual_parameter
                          | actual_parameter

adding_op                 : ‘+’
                          | ‘-’
                          | OR

array_type                : ARRAY ‘[‘ index_type_list ‘]’ OF type_denoter

array_variable            : variable_access

assignment_statement      : variable_access ASSIGNMENT_OP expression

block                     : const_def_part
                            type_def_part
                            var_decl_part
                            pf_decl_part
                            statement_part

compound_statement        : BEGIN statement_sequence END

const_def                 : IDENTIFIER ‘=’ constant

const_def_list            : const_def_list ‘;’ const_def
                          | const_def

const_def_part            : /* empty */
                          | CONST const_def_list ‘;’

constant                  : unsigned_num
                          | ‘+’ unsigned_num
                          | ‘-’ unsigned_num
                          | IDENTIFIER
                          | ‘+’ IDENTIFIER
                          | ‘-’ IDENTIFIER
                          | STRING_LITERAL

direction                 : TO
                          | DOWNTO
```

```
expression              : simple_expression
                        | simple_expression relational_op simple_expression

factor                  : variable_access
                        | unsigned_num
                        | STRING_LITERAL
                        | function_designator
                        | ‘(‘ expression ‘)’
                        | NOT factor

for_statement           : FOR IDENTIFIER ASSIGNMENT_OP expression direction
                          expression DO statement

formal_parm_list        : /* empty */
                        | ‘(‘ formal_parm_sections ‘)’

formal_parm_section     : parameter_group
                        | VAR parameter_group

formal_parm_sections    : formal_parm_sections ‘;’ formal_parm_section
                        | formal_parm_section

function_decl           : function_heading ‘;’ block ‘;’

function_designator     : IDENTIFIER
                        | IDENTIFIER actual_parameter_list

function_heading        : FUNCTION IDENTIFIER formal_parm_list ‘:’ result_type

if_branch               : statement

if_branches             : THEN if_branch
                        | THEN if_branch ELSE if_branch

if_statement            : IF expression if_branches

index_expression        : expression

index_expression_list   : index_expression_list ‘,’ index_expression
                        | index_expression

index_spec              : IDENTIFIER DOTDOT IDENTIFIER ‘:’ IDENTIFIER

index_specs             : index_specs ‘;’ index_spec
                        | index_spec
```

```
index_type                : ordinal_type

index_type_list           : index_type_list ‘,’ index_type
                          | index_type

indexed_variable          : array_variable ‘[‘ index_expression_list ‘]’

multiplying_op            : ‘*’
                          | ‘/’
                          | DIV
                          | MOD
                          | AND

new_array_type            : PACKED array_type
                          | array_type

new_fp                    : IDENTIFIER

new_fplist                : new_fplist ‘,’ new_fp
                          | new_fp

new_identifier_list       : new_identifier_list ‘,’ new_identifier
                          | new_identifier

new_ordinal_type          : subrange_type

new_type                  : new_ordinal_type
                          | new_array_type

ordinal_type              : new_ordinal_type
                          | ordinal_type_identifier

ordinal_type_identifier   : type_identifier

parameter_group           : new_fplist ‘:’ parm_type

parm_type                 : IDENTIFIER
                          | ARRAY ‘[‘ index_specs ‘]’ OF parm_type
                          | PACKED ARRAY ‘[‘ index_specs ‘]’ OF IDENTIFIER

pf_decl_part              : /* empty */
                          | pf_decls

pf_decl                   : proc_decl
                          | function_decl
```

```
pf_decls               : pf_decls pf_decl
                       | pf_decl

proc_decl              : PROCEDURE IDENTIFIER
                         formal_parm_list ‘;’ block ‘;’

procedure_statement    : IDENTIFIER
                       | IDENTIFIER actual_parameter_list

program                : PROGRAM IDENTIFIER ‘;’ block ‘.’

relational_op          : ‘=’
                       | NE
                       | ‘<‘
                       | ‘>’
                       | LE
                       | GE

repetitive_statement   : for_statement
                       | while_statement

result_type            : IDENTIFIER

sign                   : /* empty */
                       | ‘+’
                       | ‘-’

simple_expression      : sign term
                       | simple_expression adding_op term

simple_statement       : /* empty */
                       | assignment_statement
                       | procedure_statement

statement              : simple_statement
                       | structured_statement

statement_part         : compound_statement

statement_sequence     : statement_sequence ‘;’ statement
                       | statement

structured_statement   : compound_statement
                       | if_statement
                       | repetitive_statement
```

```
subrange_type           : constant DOTDOT constant

term                    : factor
                        | term multiplying_op factor

type_def                : IDENTIFIER ‘=’ type_denoter

type_def_list           : type_def_list ‘;’ type_def
                        | type_def

type_def_part           : /* empty */
                        | TYPE type_def_list ‘;’

type_denoter            : type_identifier
                        | new_type

type_identifier         : IDENTIFIER

unsigned_num            : UNSIGNED_INT
                        | UNSIGNED_REAL

var_decl                : new_identifier_list ‘:’ type_denoter

var_decl_list           : var_decl_list ‘;’ var_decl
                        | var_decl

var_decl_part           : /* empty */
                        | VAR var_decl_list ‘;’

variable_access         : IDENTIFIER
                        | indexed_variable

while_statement         : WHILE expression DO statement
```

A.2 Net Description Grammar

```
decl_section            : DECLARATIONS '{' declaration_list '}'

declaration_list        : declaration_list var_declaration
                        | var_declaration

edge_type               : TRUE
                        | FALSE

execution_step          : EXECUTION_STEP INTEGER '{' node_specs '}'

execution_steps         : execution_steps execution_step
                        | execution_step

integer_list            : integer_list ',' integer_list_value
                        | integer_list_value

integer_list_value      : INTEGER

net_description         : decl_section
                          program_section
                          optional_proc_sections

net_var_assignment      : net_variable '=' possible_values

net_variable            : NUMBER_OF_EXECUTION_STEPS
                        | NUMBER_OF_NODES
                        | BLOCK_DEPTH

netdata_section         : '{' netinfo_list execution_steps '}'

netinfo_list            : netinfo_list net_var_assignment
                        | net_var_assignment

node_data               : NODE INTEGER '{' node_entry_list '}'

node_edge_list          : node_edge_list ',' node_edge_pair
                        | node_edge_pair

node_edge_pair          : INTEGER edge_type

node_entry              : supernode_specs
                        | node_var_assignment
```

```
node_entry_list          : node_entry_list node_var_assignment
                         | node_var_assignment

node_list                : node_list node_data
                         | node_data

node_specs               : /* empty */
                         | node_list

node_var_assignment      : node_variable '=' possible_values

node_variable            : NODE_TYPE
                         | SOURCE
                         | LINE_NO
                         | HARD_DEP
                         | SOFT_DEP
                         | DATA_DEP
                         | CONTROL_DEP
                         | IO_DEP
                         | DEP_REGION
                         | REGION_HEADER_NODE
                         | CONDITION_REGION
                         | EFFERENT_REGION_DEP
                         | DEF
                         | USE

optional_proc_sections : /* empty */
                         | procedure_sections

possible_values          : STRING_LITERAL
                         | integer_list
                         | node_edge_list

procedure_section        : PROCEDURE IDENTIFIER netdata_section

procedure_sections       : procedure_sections procedure_section
                         | procedure_section

program_section          : PROGRAM IDENTIFIER netdata_section

supernode_entry          : supernode_variable '=' possible_values

supernode_entry_list     : supernode_entry_list supernode_entry
                         | supernode_entry
```

```
supernode_specs         : SUPERNODE INTEGER ‘{‘ supernode_entry_list ‘}’

supernode_variable      : CONSTITUENT_NODES
                        | ENTRY_NODES
                        | EXIT_NODES

var_declaration         : IDENTIFIER ‘,’ INTEGER ‘,’ INTEGER
```

Appendix B

Algorithms

The symbols and convenience functions described in Table B.1 are used in the algorithms presented in this section.

Symbol/Function	Representation
n, m	a single node in a net
e, f	a single execution step
N, M	an entire dependency net
S	a supernode
Q, q	a queue
CT	coalescing threshold
NDT	number of dependencies threshold
N.Esteps	the number of execution steps in net *N*
DeQ(q)	remove and return the item from the head of queue *q*
EnQ(q,x)	add item *x* to queue *q*
AddToSN(S,n)	adds node *n* to supernode *S*
RemoveFromSN(S,n)	removes node *n* from supernode *S*
SN(n)	returns the supernode to which node *n* belongs
DStr(x,y)	returns the dependency strength between nodes *x* and *y*
NDep(x,y)	returns the number of dependencies between nodes *x* and *y*
PlaceNode(n,e)	places node *n* into execution step *e*
CreateSN(e)	creates a new supernode starting in execution step *e*

Table B.1 - Algorithm symbols and convenience functions

The following convention is also used when referring to a specific element of data associated with a node:

Example: To reference the execution step number of node *n:* n.Estep

B.1 Alias Resolution

```
ResolveAliases()

    Input:none
    Output:none

for each variable declaration x in the program
    for each variable decl y that has been indicated as an alias of x
        set $x \equiv y$ and $y \equiv x$
```

B.2 Dependency Computation

ComputeDependencies()

```
Input:   none
Output:  none

for each net N
{
   for each node n previously seen in N
   {
      apply aliases to n.def and n.use sets
      for each node m already visited
      {
         Hdep = false
         Ddep = false
         IOdep = false

         if ($m.CD \neq n.CD$) then
         {
            Add m to n.SD
            Add n to m.SD
         }
         else
         {
            if ($m \in n.CD$) then
            {
               Hdep = true
            }

            if ($n.use \wedge m.def \neq \emptyset$) then
            {
               Hdep = true
               Ddep = true
            }

            if ($n.def \wedge m.use \neq \emptyset$) then
            {
               Hdep = true
               Ddep = true
            }

            if ($n.def \wedge m.def \neq \emptyset$) then
            {
               Hdep = true
               Ddep = true
            }

            if (n.Ntype = IO AND m.Ntype = IO) then
            {
               Hdep = true
               IOdep = true
            }
```

```
                if (Hdep = true) then
                {
                    if (n.DR ≠ m.DR) then
                    {
                        add n to m.Eff
                        add m to n.Eff
                    }

                    if (PathExists(n,m) = false) then
                    {
                        Add m to n.HD
                        Add n to m.HD

                        if (Ddep = true) then
                        {
                            add m to n.DD
                            add n to m.DD
                        }

                        if (IOdep = true) then
                        {
                            add m to n.IO
                            add n to m.IO
                        }
                    }
                }
            }
        }
    }
}
```

PathExists(currentNode, seenNode)

Input: currentNode ∈ node
seenNode ∈ node
Output: *true* if a path of dependence edges exists between currentNode and seenNode.

false if a path does not exist between the nodes.

```
initialize Q
reset marked nodes set

EnQ(Q,currentNode)
while (Q is not empty)
{
   n = DeQ(Q)
   if (n = seenNode) then return (true)
   else
   {
      if (n is not marked) then
      {
         mark node n
         for each node m ∈ n.HD
         {
            if (m.NN < n.NN) then EnQ(Q,m)
         }
      }
   }
}

return (false)
```

B.3 Node Migration

MigrateNodes()

```
Input:   none
Output:  none

for each net N
{
   create a new net M $\ni$ M.Esteps = N.Esteps
   for each execution step e $\in$ N
   {
      for each node n $\in$ e
      {
         AddNode(n,e,M)
      }
   }
   PlaceNode(NET_TERMINATE,M.Esteps+1)
}
```

AddNode(n, e, M)

```
Input:   n $\in$ node
         e $\in$ execution step
         M $\in$ net
Output:  none

lastSafeEstep = e
for each execution step f visited already in M
{
   for each node m $\in$ f
   {
      if (DependencyExists(m,n) = true) then
      {
         PlaceNode(n,lastSafeEstep)
         return;
      }
   }
   lastSafeEstep = f
}

PlaceNode(n,lastSafeEstep)
```

DependencyExists(m, n)

Input: $m \in$ node
$n \in$ node
Output: *true* if a dependency exists between the nodes
false if a dependency does not exist

```
if (m.DR = n.DR) then
{
   if (m ∈ n.HD) then return true
   else return false
}
else
{
   if (m.DR < n.DR) then
   {
      if (m ∈ n.DR.Eff) then return true
      else return false
   }
   else
   {
      if (n ∈ m.DR.Eff) then return true
      else return false
   }
}
```

B.4 Calculation Of Target Nodes

Target(n)

```
Input:    n ∈ node
Output:   the set of all nodes that can be executed immediately after
          node n

if (n.Ntype = IF_PREDICATE) then
{
   found_false = False
   for each node m ∈ (n.Estep + 1) ∋ m ∉ n.SD
   {
      if (n ∈ m.CD) then
      {
         Add m to Target(n)
         if ( (n,False) ∈ m.CD ) then
         {
            found_false = True
         }
         else
         {
            cond_region = m.CR
         }
      }
   }

   if (found_false = False) then
   {
      found_node = False
      for each execution step e ∋ e > n.Estep
      {
         for each node m ∈ e ∋ m ∉ n.SD
         {
            if (m.CR < cond_region) then
            {
               Add m to Target(n)
               found_node = True
            }
         }

         if (found_node = True) then return
      }
   }
}
```

```
else if (n.Ntype = LOOP_HEADER) then
{
   for each node $m \in (n.Estep + 1) \ni m \notin n.SD$
   {
      if ($n \in m.CD$) then
      {
         Add m to Target(n)
      }
   }

   found_node = False
   for each execution step $e \ni e > n.Estep + 1$
   {
      for each node $m \in e \ni m \notin n.SD$
      {
         if ($m.DR < n.DR$) then
         {
            Add m to Target(n)
            found_node = True
         }
      }

      if (found_node = True) then return
   }
}

else
{
   found_node = False
   for each execution step $e \ni e > n.Estep$
   {
      for each node $m \in e \ni m \notin n.SD$
      {
         if (m.DR ? n.DR) then
         {
            Add m to Target(n)
            found_node = True
         }
         else
         {
            Add n.RHN to Target(n)
            found_node = True
         }
      }

      if (found_node = True) then return
   }
}
```

B.5 Coalescing By Dependency Strength

The following steps are applied (in order) to each net N generated from the initial analysis stages.

1. Supernode Formation

```
initialize ToVisit_Q
for each execution step $e \in N$
{
   for each $n \in e \ni SN(n) = \emptyset$
   {
      S = CreateSN(e)
      AddToSN(S,n)
      EnQ(ToVisit_Q,n)
      while (ToVisit_Q is not empty)
      {
         n = DeQ(ToVisit_Q)
         for each node $m \in (n.Estep \cup n.Estep+1) \ni (m \neq n)$
         {
            if $(m.DR = n.DR) \wedge (m \notin n.SD) \wedge$
               $(DStr(n,m) \geq CT) \wedge (m.CR \geq n.CR) \wedge$
               $(SN(m) = \emptyset)$ then
               {
               AddToSN(SN(n),m)
               EnQ(ToVisit_Q,m)
               }
         }
      }
   }
}
```

2. Split Supernodes

```
for each execution step $e \in N$
{
   for all supernodes $S \in e$
   {
      for each node $n \in S$
      {
         for each node $m \in (n.Estep+1) \ni (m \neq n)$
         {
            if $(m.DR = n.DR) \wedge (m \notin n.SD) \wedge$
               $(DStr(n,m) \geq CT) \wedge (m.CR \geq n.CR) \wedge$
               $(SN(n) \neq SN(m))$ then
            {
               SplitSN(SN(m),m)
            }
            if (SN(m).Estep ≤ SN(n).Estep) ∧
               (SN(n) ≠ SN(m)) ∧
               (m.DR = n.DR) ∧ (m ∉ n.SD) then
            {
               SplitSN(SN(m),m)
            }
         }
      }
   }
}
```

3. Compress Net

```
nextEStep = 0
for each execution step e ∈ N
{
   if (|e| ≠ Ø) then
   {
      for each S ∈ e
      {
         S.Estep = nextEStep
      }
      nextEStep++
   }
}
```

4. Identify Entry/Exit Nodes

```
Set NET_INITIALIZE node as an ENTRY node
Set NET_TERMINATE node as an EXIT node
for each supernode S ∈ e
{
   for each node n ∈ S
   {
      for each node m ∈ Target(n)
      {
         if (m ∉ S) then
         {
            set m as an ENTRY node
            set n as an EXIT node
         }
      }
   }
}
```

SplitSN(S, n)

```
Input:   S ∈ supernode
         n ∈ node
Output:  None

Initialize ToVisit_Q
T = CreateSN(n.Estep)
EnQ(ToVisit_Q,n)
while (ToVisit_Q is not empty)
{
   m = DeQ(ToVisit_Q)
   RemoveFromSN(S,n)
   AddToSN(T,m)
   for each node n ∈ (m.Estep+1)
   {
      if n ∈ S then EnQ(ToVisit_Q,n)
   }
}
```

B.6 Coalescing By Number Of Dependencies

The following steps are applied (in order) to each net N generated from the initial analysis stages.

1. Supernode Formation

```
initialize ToVisit_Q
for each execution step $e \in N$
{
   for each $n \in e \ni SN(n) = \emptyset$
   {
      S = CreateSN(e)
      AddToSN(S,n)
      EnQ(ToVisit_Q,n)
      while (ToVisit_Q is not empty)
      {
         n = DeQ(ToVisit_Q)
         for each node $m \in (n.Estep \cup n.Estep+1) \ni (m \neq n)$
         {
            if $(m.DR = n.DR) \wedge (m \notin n.SD) \wedge$
               $(NDep(n,m) \geq NDT) \wedge (m.CR \geq n.CR) \wedge$
               $(SN(m) = \emptyset)$ then
               {
               AddToSN(SN(n),m)
               EnQ(ToVisit_Q,m)
               }
         }
      }
   }
}
```

2. Split Supernodes

```
for each execution step $e \in N$
{
   for all supernodes $S \in e$
   {
      for each node $n \in S$
      {
         for each node $m \in (n.Estep+1) \ni (m \neq n)$
         {
            if $(m.DR = n.DR) \wedge (m \notin n.SD) \wedge$
               $(NDep(n,m) \geq NDT) \wedge (m.CR \geq n.CR) \wedge$
               $(SN(n) \neq SN(m))$ then
            {
               SplitSN(SN(m),m)
            }
```

```
                if (SN(m).Estep ≤ SN(n).Estep) ∧
                   (SN(n) ≠ SN(m)) ∧
                   (m.DR = n.DR) ∧ (m ∉ n.SD) then
                {
                   SplitSN(SN(m),m)
                }
             }
          }
       }
    }
```

3. Compress Net

4. Identify Entry/Exit Nodes

B.7 Coalescing For Maximal Node Parallelism

The following steps are applied (in order) to each net N generated from the initial analysis stages.

1. <u>Supernode Formation</u>

```
initialize newSN_Q, nextNode_Q
EnQ(newSN_Q,NET_INITIALIZE)
while (newSN_Q is not empty)
{
   n = DeQ(newSN_Q)
   S = CreateSN(n.Estep)
   Set n as an ENTRY node
   lastNode = n
   EnQ(nextNode_Q,n)

   while (nextNode_Q is not empty)
   {
      n = DeQ(nextNode_Q)
      if ($n \in$ newSN_Q) then
      {
         continue
      }

      if ($|$Target(n)$|$ > 1) then
      {
         for each node $m \in$ Target(n) $\ni$ ($m \neq$ n.RHN)
         {
            EnQ(newSN_Q,m)
         }
         lastNode = n
         AddToSN(S,n)
      }
      else
      {
         fanin = 0
         for each node $m \in$ N
         {
            if ($n \in$ Target(m)) then fanin++
         }

         if (fanin > 1) then
         {
            if ($|$S$|$ = 0) then
            {
               AddToSN(S,n)
               lastNode = n
            }
            else EnQ(newSN_Q,n)
         }
         else
         {
            AddToSN(S,n)
            lastNode = n
            EnQ(nextNode_Q,Target(n))
         }
      }
   }
   mark lastNode as an EXIT node
}
```

2. Compress Net

Appendix C

Sample Files

C.1 Sample Net Description File

```
DECLARATIONS {

     b, 0, 0
     a, 0, 1
     xxsum, 0, 2
     xysum, 0, 3
     ysum, 0, 4
     xsum, 0, 5
     yi, 0, 6
     xi, 0, 7
     i, 0, 8

}

PROGRAM LeastSquares {

     NUMBER_OF_EXECUTION_STEPS = 9
     NUMBER_OF_NODES = 15
     BLOCK_DEPTH = 0

     EXECUTION_STEP 0 {

        NODE 0 {
           NODE_TYPE = 0
           SOURCE = "NET_INITIALIZE"
           HARD_DEP =
           DEP_REGION = 0
           REGION_HEADER_NODE = 0
           CONDITION_REGION = 0
        }

        SUPERNODE 0 {
           CONSTITUENT_NODES = 0, 1, 2, 3, 4
           ENTRY_NODES = 0
           EXIT_NODES = 1, 2, 3, 4
        }

     }

     EXECUTION_STEP 1 {
```

```
NODE 1 {
   NODE_TYPE = 2
   SOURCE = “xsum := 0”
   LINE_NO = 14
   HARD_DEP = 0, 7
   DATA_DEP = 7
   CONTROL_DEP = 0
   DEP_REGION = 0
   REGION_HEADER_NODE = 0
   CONDITION_REGION = 0
   DEF = 5
}

NODE 2 {
   NODE_TYPE = 2
   SOURCE = “ysum := 0”
   LINE_NO = 15
   HARD_DEP = 0, 8
   DATA_DEP = 8
   CONTROL_DEP = 0
   DEP_REGION = 0
   REGION_HEADER_NODE = 0
   CONDITION_REGION = 0
   DEF = 4
}

NODE 3 {
   NODE_TYPE = 2
   SOURCE = “xysum := 0”
   LINE_NO = 16
   HARD_DEP = 0, 9
   DATA_DEP = 9
   CONTROL_DEP = 0
   DEP_REGION = 0
   REGION_HEADER_NODE = 0
   CONDITION_REGION = 0
   DEF = 3
}

NODE 4 {
   NODE_TYPE = 2
   SOURCE = “xxsum := 0”
   LINE_NO = 17
   HARD_DEP = 0, 10
   DATA_DEP = 10
   CONTROL_DEP = 0
   DEP_REGION = 0
   REGION_HEADER_NODE = 0
   CONDITION_REGION = 0
   DEF = 2
}
```

```
   SUPERNODE 1 {
      CONSTITUENT_NODES = 5, 6
      ENTRY_NODES = 5
      EXIT_NODES = 6
   }

}

EXECUTION_STEP 2 {

   NODE 5 {
      NODE_TYPE = 5
      SOURCE = “FOR i := 1 TO n”
      LINE_NO = 18
      HARD_DEP = 0, 6
      CONTROL_DEP = 0, 6
      DEP_REGION = 1
      REGION_HEADER_NODE = 5
      CONDITION_REGION = 0
      EFFERENT_REGION_DEP = 1, 2, 3, 4, 11
      DEF = 8
   }

   SUPERNODE 2 {
      CONSTITUENT_NODES = 7
      ENTRY_NODES = 7
      EXIT_NODES = 7
   }

   SUPERNODE 3 {
      CONSTITUENT_NODES = 8
      ENTRY_NODES = 8
      EXIT_NODES = 8
   }

   SUPERNODE 4 {
      CONSTITUENT_NODES = 9
      ENTRY_NODES = 9
      EXIT_NODES = 9
   }

   SUPERNODE 5 {
      CONSTITUENT_NODES = 10
      ENTRY_NODES = 10
      EXIT_NODES = 10
   }

}

EXECUTION_STEP 3 {

   NODE 6 {
      NODE_TYPE = 6
      SOURCE = “READ(xi, yi)”
      LINE_NO = 20
      HARD_DEP = 5, 7, 8, 9, 10
      DATA_DEP = 7, 8, 9, 10
      CONTROL_DEP = 5
      DEP_REGION = 1
      REGION_HEADER_NODE = 5
      CONDITION_REGION = 0
      DEF = 6, 7
   }

   SUPERNODE 6 {
      CONSTITUENT_NODES = 11
      ENTRY_NODES = 11
      EXIT_NODES = 11
   }

}

EXECUTION_STEP 4 {

   NODE 7 {
      NODE_TYPE = 2
      SOURCE = "xsum := xsum + xi"
      LINE_NO = 21
      HARD_DEP = 1, 6, 11
      DATA_DEP = 1, 6, 11
      DEP_REGION = 1
      REGION_HEADER_NODE = 5
      CONDITION_REGION = 0
      DEF = 5
      USE = 5, 7
   }

   NODE 8 {
      NODE_TYPE = 2
      SOURCE = "ysum := ysum + yi"
      LINE_NO = 22
      HARD_DEP = 2, 6, 11
      DATA_DEP = 2, 6, 11
      DEP_REGION = 1
      REGION_HEADER_NODE = 5
      CONDITION_REGION = 0
      DEF = 4
      USE = 4, 6
   }

   NODE 9 {
      NODE_TYPE = 2
      SOURCE = "xysum := xysum + xi * yi"
      LINE_NO = 23
      HARD_DEP = 3, 6, 11
      DATA_DEP = 3, 6, 11
      DEP_REGION = 1
      REGION_HEADER_NODE = 5
```

```
        CONDITION_REGION = 0
        DEF = 3
        USE = 3, 6, 7
    }

    NODE 10 {
        NODE_TYPE = 2
        SOURCE = "xxsum := xxsum + xi * xi"
        LINE_NO = 24
        HARD_DEP = 4, 6, 11
        DATA_DEP = 4, 6, 11
        DEP_REGION = 1
        REGION_HEADER_NODE = 5
        CONDITION_REGION = 0
        DEF = 2
        USE = 2, 7
    }

    SUPERNODE 7 {
        CONSTITUENT_NODES = 12
        ENTRY_NODES = 12
        EXIT_NODES = 12
    }

}

EXECUTION_STEP 5 {

    NODE 11 {
        NODE_TYPE = 2
        SOURCE = "b := (n*xysum - xsum*ysum)/(n*xxsum - xsum*xsum)"
        LINE_NO = 26
        HARD_DEP = 7, 8, 9, 10
        DATA_DEP = 7, 8, 9, 10
        DEP_REGION = 0
        REGION_HEADER_NODE = 0
        CONDITION_REGION = 0
        DEF = 0
        USE = 2, 3, 4, 5
    }

    SUPERNODE 8 {
        CONSTITUENT_NODES = 13
        ENTRY_NODES = 13
        EXIT_NODES = 13
    }

}

EXECUTION_STEP 6 {

    NODE 12 {
        NODE_TYPE = 2
        SOURCE = "a := (ysum - b*xsum)/n"
```

```
            LINE_NO = 27
            HARD_DEP = 11
            DATA_DEP = 11
            DEP_REGION = 0
            REGION_HEADER_NODE = 0
            CONDITION_REGION = 0
            DEF = 1
            USE = 0, 4, 5
         }

         SUPERNODE 9 {
            CONSTITUENT_NODES = 14
            ENTRY_NODES = 14
            EXIT_NODES = 14
         }

      }

      EXECUTION_STEP 7 {

         NODE 13 {
            NODE_TYPE = 6
            SOURCE = "writeln('values of a and b are ',a,b)"
            LINE_NO = 29
            HARD_DEP = 12
            DATA_DEP = 12
            DEP_REGION = 0
            REGION_HEADER_NODE = 0
            CONDITION_REGION = 0
            USE = 0, 1
         }

      }

      EXECUTION_STEP 8 {

         NODE 14 {
            NODE_TYPE = 1
            SOURCE = "NET_TERMINATE"
            DEP_REGION = 0
            REGION_HEADER_NODE = 0
            CONDITION_REGION = 0
         }

      }

   }
```

C.2 Sample Node Coalescing Parameter File

```
COALESCING_THRESHOLD 3.0
PSEUDO_CONTROL_T     1.3
PSEUDO_CONTROL_F     .45
PSEUDO_CONTROL_IO    1.5
```

C.3 Sample Program analysis

```
      PROGRAM analysis;

      { This program reads 61 population values and determines the
        greatest percentage increase in population. }

      VAR
         percent, bestpercent : REAL;
         i, year1, year2, bestyear : INTEGER;
         oldyear, oldpopulation : INTEGER;
         newyear, newpopulation : INTEGER;

      BEGIN
1        readln (oldyear,oldpopulation);
2        bestpercent := 0.0;
3        FOR i := 1 TO 60 DO
            BEGIN
4              readln(newyear,newpopulation);
5              percent := (newpopulation - oldpopulation) * 100.0
                              / oldpopulation;
6              IF (percent > bestpercent) OR (i = 1) THEN
                  BEGIN
7                    bestpercent := percent;
8                    bestyear := newyear
                  END;
9              oldyear := newyear;
10             oldpopulation := newpopulation
            END;
11       year1 := bestyear - 1;
12       year2 := bestyear;
13       writeln('Greatest percent increase occurred between ',year1,
                 ' and ', year2)
      END.
```

C.4 Sample Program `LeastSquares`

```
PROGRAM LeastSquares;

{ Determines the coefficients of the linear equation y = ax + b
  that fit a regression of points }

CONST n = 10;

VAR
   i: INTEGER;
   xi,yi,xsum,ysum,xysum,xxsum,a,b : REAL;

BEGIN

   xsum := 0;
   ysum := 0;
   xysum := 0;
   xxsum := 0;
   FOR i := 1 TO n DO
      BEGIN
         READ(xi, yi);
         xsum := xsum + xi;
         ysum := ysum + yi;
         xysum := xysum + xi * yi;
         xxsum := xxsum + xi * xi;
      END;
   b := (n*xysum - xsum*ysum)/(n*xxsum - xsum*xsum);
   a := (ysum - b*xsum)/n;

   writeln(‘values of a and b are ’,a,b)

END.
```

C.5 Sample Program graphics

```
PROGRAM graphics;

CONST
     screenwidth = 60;
     screenht = 26;

VAR
     cursorx,cursory : INTEGER;
     screen : ARRAY[0..screenwidth,0..screenht] OF CHAR;
     x,y,length,dn : INTEGER;
     forever : BOOLEAN;

     PROCEDURE fail(x,y : INTEGER);
     BEGIN
        WRITELN(x,', ',y,' is off the screen')
     END;



     FUNCTION abs(x : INTEGER): INTEGER;
     BEGIN
        IF (x < 0) THEN
           abs := x * -1
        ELSE
           abs := x
     END;



     PROCEDURE lineby(xdif,ydif : INTEGER);

        CONST star = '*';

        VAR
           xmax : BOOLEAN;
           xsign, ysign,i,maxval,finalx,finaly : INTEGER;

     BEGIN
        finalx := xdif + cursorx;
        IF (finalx > screenwidth) OR (finalx < 0) THEN
           fail(finalx,finaly);

        finaly := ydif + cursory;
        IF (finaly > screenht) OR (finaly < 0) THEN
           fail(finalx,finaly);

        screen[cursorx,cursory] := star;

        IF (xdif <> 0) OR (ydif <> 0) THEN
        BEGIN
```

```
      xmax := abs(xdif) > abs(ydif);
      IF xmax THEN
         slope := abs(ydif/xdif)
      ELSE
         slope := abs(xdif/ydif);

      IF xdif > 0 THEN
         xsign := 1
      ELSE
         xsign := -1;

      IF ydif > 0 THEN
         ysign := 1
      ELSE
         ysign := -1;

      sum := 0.5;

      IF xmax THEN
         maxval := abs(xdif)
      ELSE
         maxval := abs(ydif);

      FOR i := 1 TO maxval DO
      BEGIN
         sum := sum + slope;
         IF sum > 1.0 THEN
         BEGIN
            IF xmax THEN
               cursory := cursory + ysign
            ELSE
               cursorx := cursorx + xsign
            sum := sum - 1
         END;

         IF xmax THEN
            cursorx := cursorx + xsign
         ELSE
            cursory := cursory + ysign

         screen[cursorx,cursory] := star;
      END
   END
END


PROCEDURE lineto(u,v : INTEGER);

   VAR xdif,ydif : INTEGER;

BEGIN
   xdif := u-cursorx;
   ydif := v-cursory;
   lineby(xdif,ydif)
END;



PROCEDURE moveby(a,b : INTEGER);

   VAR error : BOOLEAN;

BEGIN
   IF a < 0 THEN
      error := -a > cursorx
   ELSE
      error := screenwidth-a < cursorx;

   IF NOT error THEN
      IF b < 0 THEN
         error := -b > cursory
      ELSE
         error := screenht-b < cursory;

   IF error THEN fail(cursorx+a,cursory+b);

   cursorx := cursorx + a;
   cursory := cursory + b

END;



PROCEDURE moveto(x,y : INTEGER);
BEGIN
   IF (x>screenwidth) OR (y>screenht) OR (x<0) OR (y<0) THEN
      fail(x,y)

   cursorx := x;
   cursory := y

END;



PROCEDURE insertspaces;

   VAR row,col : INTEGER;

BEGIN
   FOR col := 0 TO screenwidth DO
      FOR row := 0 TO screenht DO
         screen[row,col] := space
END;
```

```
    PROCEDURE drawgraph;

       VAR row,col : INTEGER;

    BEGIN
       FOR row := 0 TO screenht DO
       BEGIN
          FOR col := 0 TO screenwidth DO
             write(screen[row,col]);
          writeln
       END
    END;

BEGIN
1     insertspaces;
2     moveto(screenwidth DIV 2,screenht DIV 2);
3     x := screenwidth DIV 2;
4     y := screenht DIV 2;
5     dn := 1;
6     forever := True;
7     length := 0;

8     WHILE forever DO
      BEGIN
9        IF (dn = 1) THEN
            BEGIN
10             dn := 2;
11             x := x+length;
12             y := y+length
            END;
13       ELSE IF (dn = 2) THEN
            BEGIN
14             dn := 3;
15             x := x-length;
16             y := y+length
            END;
17       ELSE IF (dn = 3) THEN
            BEGIN
18             dn := 4;
19             x := x-length;
20             y := y-length
            END;
21       ELSE IF (dn = 4) THEN
            BEGIN
22             dn := 1;
23             x := x+length;
24             y := y-length
            END

25       lineto(x,y);
26       length := length + 1
      END

END.
```